Efficient Indexing for Diverse Query Results

Lu Li Chee-Yong Chan
Department of Computer Science
School of Computing
National University of Singapore
{lilu0355, chancy}@comp.nus.edu.sg

ABSTRACT

This paper examines the problem of computing diverse query results which is useful for browsing search results in online shopping applications. The search results are diversified wrt a sequence of output attributes (termed d-order) where an attribute that appears earlier in the d-order has higher priority for diversification. We present a new indexing technique, D-Index, to efficiently compute diverse query results for queries with static or dynamic d-orders. Our performance evaluation demonstrates that our D-Index outperforms the state-of-the-art techniques developed for queries with static or dynamic d-orders.

1. INTRODUCTION

Consider a user who is shopping online for a new laptop from a website which can display a result table consisting of up to 20 laptops that match the user's specification. As the number of matching results is typically much larger than number of display records, it is useful to return a diverse set of results for the user to browse. For example, instead of showing the user 20 laptops from only two brands (say Lenovo and Acer), it would be more interesting to show results covering a more diverse range of brands (e.g., Lenovo, Acer, Dell, HP, Asus, Samsung). If Lenovo and Acer are indeed the only two brands of laptops that satisfy the user's query, then it would be better to show a more "balanced" distribution of the 20 displayed laptops; for example, showing 10 laptops from each of Lenovo and Acer is better than showing 18 laptops from Lenovo and 2 laptops from Acer. Similarly, if the user is interested only in laptops from Dell, then it would be more interesting to show a diverse range of Dell laptops with different screen sizes instead of showing all Dell laptops with the same screen size.

In general, the query results can be diversified wrt a sequence of attributes, say (brand, screen size, ...), referred to as a *d-order*, where the intention is to first diversify the results with as many different brands as possible, and for records that belong to the same brand, we diversify them with as many different screen sizes as possible, and so on. Thus, a d-order determines a priority order for diversifying the query results, where the first attribute has higher priority to diversify than the second attribute, and so on.

Vee, et al. [8] were the first to study the problem of computing diverse query results. They formally define the notion of query result diversity and show that existing score based techniques are inadequate to guarantee diverse query results. They also propose an inverted-list based approach to evaluate such queries. However, their work addresses only *static diversity queries (SDQ)*, where the query results are diversified wrt a static, pre-defined d-order. Clearly, it would be useful to allow users to customize their diversification preference. For example, Alice might be more interested to diversify the results wrt laptop color first, followed by brand, whereas Bob might be more interested to diversify the results wrt the number of CPU cores first, followed by battery life and screen size.

In this paper, we examine the more general problem of evaluating *dynamic diversity queries (DDQ)* where the query results are diversified wrt a user specified d-order. A DDQ can be expressed by the following extended SQL syntax: "SELECT ... FROM R WHERE ... DIVERSIFY BY $D_1, \cdots, D_n$ LIMIT k" which retrieves a diverse set of at most k matching records from a relation R such that the records are diversified wrt a d-order $(D_1, \cdots, D_n)$. The attributes in the SELECT clause must contain all the attributes in the DIVERSIFY BY clause.

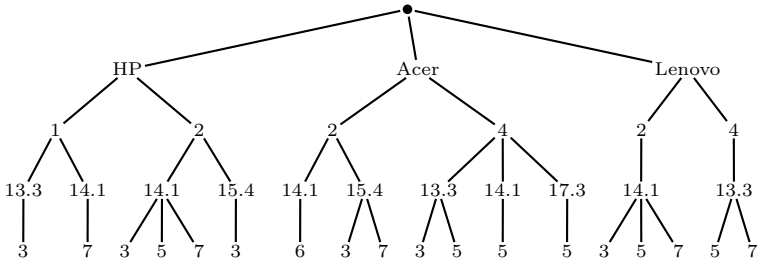
Our paper makes three key contributions. First, we show that extending existing techniques designed for SDQs [8] to evaluate DDQs is inefficient (Section 4). Second, we introduce a novel approach for evaluating diversity queries that is based on the concept of computing a *core cover of a query* (Section 5.1). Based on this concept, we design a new index method, D-Index, and introduce two index variants, namely, `D-tree` and `D`$^+$`-tree` (Sections 5 and 6). Third, we demonstrate with an experimental evaluation, which is based on a PostgreSQL implementation, that our proposed D-Index technique consistently outperforms [8] for both SDQs as well as DDQs (Section 7).

In this paper, we use Q to denote a diversity query on a relation R with d-order $\delta = (D_1, \cdots, D_m)$, a set of (possibly empty) selection predicate attributes θ, and a limit value of k. Our running example data for R is shown in Figure 1(a): the attributes Brand, #Core, ScrnSze, BatLife, and Color represent, respectively, laptop brand (B), number of CPU cores (C), screen size in inches (SS), battery life in hours (BL), and laptop color (LC).

Proceedings of the VLDB Endowment, Vol. 6, No. 9

RID	Brand	#Core	ScrnSze	BatLife	Color
1	HP	1	13.3	3	Red
2	HP	1	14.1	7	White
3	HP	2	14.1	3	Silver
4	HP	2	14.1	5	Silver
5	HP	2	13.3	7	Black
6	HP	2	15.4	3	Red
7	Acer	2	14.1	6	White
8	Acer	2	15.4	3	Silver
9	Acer	2	15.4	7	Red
10	Acer	4	13.3	3	Black
11	Acer	4	13.3	5	Black
12	Acer	4	14.1	5	Red
13	Acer	4	17.3	5	Black
14	Lenovo	2	14.1	3	White
15	Lenovo	2	14.1	5	Silver
16	Lenovo	2	14.1	7	Black
17	Lenovo	4	13.3	5	Black
18	Lenovo	4	13.3	7	White

(a)



(b)

Figure 1: Running Example (a) Relation R (b) D-index on R with key (Brand, #Core, ScrnSze, BatLife)

RID	B	C	SS
1	HP	1	13.3
4	HP	2	14.1
6	HP	2	15.4
15	Lenovo	2	14.1

(a) S_1

RID	B	C	SS
1	HP	1	13.3
4	HP	2	14.1
14	Lenovo	2	14.1
15	Lenovo	2	14.1

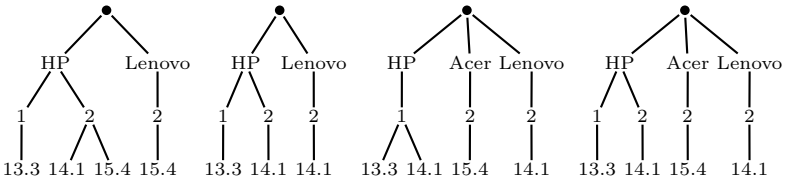
(b) S_2

RID	B	C	SS
1	HP	1	13.3
2	HP	1	14.1
8	Acer	2	15.4
15	Lenovo	2	14.1

(c) S_3

RID	B	C	SS
1	HP	1	13.3
4	HP	1	14.1
8	Acer	2	15.4
15	Lenovo	2	14.1

(d) S_4



(e) T_1 (f) T_2 (g) T_3 (h) T_4

Figure 2: Diverse Query Results, d-order δ = (Brand, #Core, ScrnSze)

2. DIVERSE QUERY RESULTS

In this section, we present the definition of diverse query results used in this paper. Our definition is based on that from [8], where a query result is diversified wrt a sequence of attributes $\delta = (D_1, \cdots, D_m)$, referred to as a *d-order*. Essentially, δ specifies a priority order for diversifying the query results with D_i having a higher priority than D_{i+1} such that we maximize the domain values shown for D_i before D_{i+1}. The goal is to maximize the diversity of the attribute domain values shown as well as "balance" the number of records for each attribute value.

Example 2.1 *Consider a query Q on R (Fig. 1) with $k = 4$ and a selection predicate "#Core $\leq$ 2". Figs. 2(a)-(d) show four possible result sets (S_1 to S_4) for Q, where only the attribute values for RID, B, C, and SS are shown. If the d-order for Q is $\delta = (B, C, SS)$, we can organize each result set S_i using a trie T_i (wrt δ) as depicted in Figs. 2(e)-(g) which provides a more visual and convenient representation for comparing result diversity. Observe that T_1 and T_2 are equally diverse wrt the brand attribute (each has two distinct brand values), but S_2 is more balanced than S_1 because S_2 has two records for each brand value, whereas S_1 has three records for HP brand and one record for Lenovo brand. However, compared to T_3, both T_1 and T_2 are less diverse wrt the brand attribute. Finally, we note that T_4 is more diverse than T_3: while both are equally diverse wrt the brand attribute (each has three brand values), T_4 is more diverse wrt the #core attribute because T_4 has two distinct #core values for its two records with HP brand, whereas T_3 has only one distinct #core value for its two records with HP brand.* □

In the following, we formalize the above intuition of diverse query results.

Definition 2.1 (attribute ordering) *An attribute ordering of a relation R is a sequence of attributes $(A_1, \cdots, A_n)$, where each A_i is a distinct attribute of R.*

Note that an attribute ordering does not necessarily include all the attributes of R.

Consider an attribute ordering $\alpha = (A_1, \cdots, A_n)$ of R. We use α_i to denote the length-i, $i \in [0, n]$, prefix of α; i.e., $\alpha_i = (A_1, \cdots, A_i)$. We refer to each α_i as a *α-prefix*.

Definition 2.2 (α-tuple, α-prefix tuple) *A tuple t is defined to be an α-tuple if $t \in \pi_\alpha(R)$ for some attribute ordering α. We say that t is an α-prefix tuple if t is an α_i-tuple for some prefix α_i of α.*

Definition 2.3 (matching α_i-tuple) *An α_i-tuple t, $i \in [1, n]$ is defined to be a matching tuple for Q if all the attributes in the selection predicates (i.e., θ) occur in α_i and t satisfies all the selection predicates of Q.*

Note that it is not necessary for a matching tuple to contain all the d-order attributes or all the attributes projected by the query.

Definition 2.4 (tuple cover) *Given a α-tuple t_a and a β-tuple t_b, we say that t_a covers t_b (or t_b is covered by t_a) if $\alpha \subseteq \beta$ and $t_a.A_i = t_b.A_i$ for each attribute $A_i \in \alpha$. We say that a tuple t covers a set of tuples S if t covers each $t' \in S$.*

Let $S \subseteq R$ be a result set for a diversity query Q on relation R wrt d-order δ, and T be the trie representation of S (wrt δ). Each node v in T corresponds to a unique δ-prefix tuple, which we denote by $ptup_\delta(v)$. For example, in Fig. 2(f), if v refers to the rightmost leaf node in T_2, we have $ptup_\delta(v) = (Lenovo, 2, 15.4)$.

Given a node v in T, we use T_v to denote the subtrie rooted at v representing the subset of records $S(v) \subseteq S$; i.e., $S(v)$ is the set of records contained in T_v. For example, in Fig. 2(f), if v refers to the node labeled "HP" in T_2, then $S(v)$ contains two records with RID values of 1 and 4.

Consider a subtrie T_v where v has c child nodes, $v_1, \cdots, v_c$. As a measure of the diversity of $S(v)$, define the metric

$$F(S(v)) = c|S(v)| - \sigma$$

where σ is the standard deviation of the set $\{|S(v_1)|, \cdots, |S(v_c)|\}$.

To understand why the above metric is meaningful for comparing result set diversity, consider a query Q to retrieve a result set of k tuples from relation R wrt d-order δ. Consider the trie representations, T_1 and T_2, of two possible result sets, $S_1, S_2 \subseteq R$, where $|S_1| = |S_2| = k$. Let $F(S_1) = c_1 k - \sigma_1$ and $F(S_2) = c_2 k - \sigma_2$. If S_1 is more diverse than S_2, then either (1) the root node of T_1 has more child nodes than that of T_2 (i.e., $c_1 > c_2$), or (2) the root nodes of both T_1 and T_2 have the same number of child nodes, but the child subtrees in T_1 are more balanced than those in T_2 (i.e., $c_1 = c_2$ and $\sigma_1 < \sigma_2$). Effectively, $F(S_1)$ is larger than $F(S_2)$ if S_1 is more diverse than S_2.

In other words, given a result set $S \subseteq R$ of Q, if for every node v in the trie representation of S, $F(S(v))$ can not be further increased (by replacing some records in $S(v)$ by an equal number of some other records from $R - S$ that are covered by $ptup_\delta(v)$), then the diversity of S can not be increased (without increasing the cardinality of S), and we conclude that S is a diverse result set of cardinality k. Thus, we can define a diverse result set S in terms of maximizing $F(v)$ for each node v in the trie representation of S.

Definition 2.5 (diverse result set) *Let T denote the trie representation of a result set $S \subseteq R$ for a diversity query Q on R wrt d-order δ. Let T_v denote a subtrie of T rooted at v. We define S to be diverse wrt $ptup_\delta(v)$ if $F(S(v))$ is maximized over all sets $S' \subseteq R$ that are covered by $ptup_\delta(v)$ such that $|S'| = |S(v)|$. We define S to be a diverse result set for Q if S is diverse wrt every δ-prefix tuple in S.*

Example 2.2 *Consider the trie T_4 in Fig. 2(h). Let v_0 denote the root node of T_4, and v_1 denote the node in T_4 with $ptup_\delta(v_1) = (HP)$. We have $F(S_4(v_0)) = 12 - \sqrt{2}/3$ and $F(S_4(v_1)) = 4$. S_4 is a diverse result set for Q following the definition: S_4 is diverse wrt $ptup_\delta(v_0)$ since there are only three brand values in R and v_0 has three child nodes; S_4 is diverse wrt $ptup_\delta(v_1)$ since $|S_4(v_1)| = 2$ and v_1 has two child nodes; and for each of the remaining nodes v in T_4, S_4 is diverse wrt $ptup_\delta(v)$ since $|S_4(v)| = 1$. On the other hand, T_1 in Fig. 2(e) is not a diverse result set because S_1 is not diverse wrt $ptup_\delta(v_0)$ where v_0 is the root node of T_1: $F(S_1(v_0))$ can be further increased by making the child subtrees of v_0 more balanced by replacing RID6 with RID14 to obtain T_2 in Fig. 2(f).* □

Note that our definition of diverse result set is equivalent to one in [8] in that a set is a diverse result set under our definition if and only if it is also a diverse result set under the definition in [8]. We have chosen to present the definition in terms of the metric $F()$ as we believe that it captures more closely the intuition behind the diversity definition. We should emphasize that our contribution is not on the definition of diverse query result but on the efficient evaluation of diversity queries.

3. RELATED WORK

Query result diversification. Search result diversification is an active research area that aims to increase user satisfication in web search and recommender systems (e.g., [1, 4]). The area can be broadly classified into *content-based diversification* (e.g., [9, 7, 2]) which aims to reduce information redundancy in search results, and *intent-based diversification* (e.g., [10, 11, 3, 6]) which aims to provide search results that cover as many facets of the query as possible to deal with ambiguous queries.

Our work on DDQ evaluation falls under content-based diversification. We adopt the diversity definition from [8] which is intuitive to use and only requires an explicit specification of attribute ordering for diversification. In contrast, many other diversity definitions require assigning a diversity score to each record, which could be hard to interpret, or require specifying a distance function to measure the dissimilarity between a pair of records.

Static diversity queries. The work that is the most related to ours is the paper by Vee et al. [8] which introduced the problem of evaluating SDQs. They showed that existing score based techniques are inadequate for the problem and proposed two indexing methods, `OnePass` and `Probe`.

To evaluate SDQs on a relation R, `OnePass` builds an inverted-list index I_j for each attribute A_j in R, where each posting list in I_j is organized using a B$^+$-tree with a predetermined d-order, $\alpha = (A_1, \cdots, A_n)$, which consists of all the attributes in R, as the index key. Thus, all the B$^+$-trees in `OnePass` use α as the index key. The B$^+$-trees are compressed by replacing each key attribute value with a Dewey encoded value (e.g., replace "Acer" by the value 0). Given a SDQ Q with a selection predicate "$A_j = v$", `OnePass` evaluates Q by an index scan on the B$^+$-tree corresponding to the value v in I_j. A run-time, main-memory trie structure T is used to organize the retrieved index key values such that each root-to-leaf path in T represents a retrieved α-tuple. Since the index key and query's d-order are both the same (i.e., α) for SDQs, the B$^+$-tree index scan ensures that the retrieved key values are inserted into T "sequentially" by extending T with a rightmost path. This important property enables `OnePass` to conveniently detect when there is a sufficient number of α-tuples in a subtrie to form a diverse result set so that the B$^+$-tree index scan can skip to retrieve tuples for another subtrie in T. As an example, suppose that $\alpha = (A, B, C, D)$ and after inserting a newly retrieved tuple (a_1, b_1, c_1, d_1) into T, `OnePass` detects that the subtrie rooted at (a_1, b_1) has sufficient number of tuples; in this case, the index scan will skip to search for index keys greater than $(a_1, b_1, c_\infty, d_\infty)$, where c_∞ and d_∞ represent the largest domain values for attributes C and D, respectively.

To deal with multiple selection predicates on different attributes, `OnePass` invokes a B$^+$-tree index scan for each of the selection attributes and uses an appropriate merge operation to combine the index keys retrieved from the multiple index scans.

`Probe` is a variant of `OnePass` that performs a bi-directional B$^+$-tree index scan instead of the single forward scan adopted in `OnePass`. The goal of `Probe` is to reduce the number of *useless* retrieved tuples, which are tuples that are retrieved into T but are later replaced by other tuples. However, `Probe` incurs more random I/Os due to its bi-directional scan and the experimental results in [8] indicate that both `OnePass` and `Probe` performed similarly.

4. CHALLENGES FOR DYNAMIC QUERIES

To motivate the need for a new approach to evaluate D-DQs, we argue that although existing techniques for SDQs [8] can be extended to support DDQs, their performance is expected to be poor due to the need to scan a significant portion of the index. This is validated by our experimental results in Section 7.

Let us first consider how to extend the basic technique, `OnePass` [8], to form a new variant, termed `OnePass`^{D}, for evaluating DDQs. To make the discussion concrete, suppose that the B$^+$-trees in `OnePass`^{D} have index key $\alpha = (A, B, C, D, E)$ and we are using `OnePass`^{D} to evaluate a D-DQ with a d-order of $\delta = (D, E)$ and a selection predicate "$A = a_1$". Similar to `OnePass`, `OnePass`^{D} performs an index scan on the B$^+$-tree corresponding to the value a_1 in the inverted-list index I_a for attribute A. Each retrieved α-tuple from the index scan is converted to a δ-tuple to update the main-memory trie T. Due to the difference between α and δ, there are two extensions required for `OnePass`^{D} to work correctly. First, the tuples inserted into T are now in a "random" instead of a "sequential" order (e.g., the index scan returns $(a_1, b_1, c_1, d_2, e_2)$ followed by $(a_1, b_1, c_2, d_1, e_1)$, where $d_1 < d_2$). Thus, the simple scheme adopted in `OnePass` for detecting when there are sufficient tuples in a subtrie no longer works due to this random order and a more sophisticated detection scheme is required. Second, the Dewey encoding scheme used for compressing index keys does not work correctly when the α-tuples are mapped to δ-tuples (to update T) as the same attribute value could have different Dewey encodings. The second extension is trivial to fix (encode each attribute value with a unique value), but the first extension is more intricate (Section 5.5).

Although `OnePass`^{D} can work correctly to evaluate D-DQs, its performance could be very inefficient as it might need to scan the entire index. Continuing with the example, suppose that after updating T with a newly retrieved tuple $(a_1, b_1, c_1, d_1, e_1)$, `OnePass`^{D} detects that the subtrie rooted at (d_1, e_1) has sufficient number of tuples. However, `OnePass`^{D} cannot efficiently skip to search for the next value after (d_1, e_1) as the B and C attributes preceding D are not part of the search attributes. Hence, in the worst case, no index skip operation is possible in the `OnePass`^{D} approach. For similar reasons, `Probe` could be extended to correctly evaluate DDQs but would perform even worse than `OnePass`^{D} as the extended `Probe` would still incur random I/Os for its bi-directional scan but without the benefit of reducing useless tuple retrievals due to the absence of index skip operations.

5. OUR APPROACH

In this section, we present the key ideas behind our approach of evaluating diversity queries.

5.1 Core Cover

Our approach for computing diverse query results is based on the concept of computing a *core cover* for a query.

Definition 5.1 (core cover) *A set of δ-prefix tuples $C = \{t_1, \cdots, t_\ell\}$, $\ell \in [1, k]$, is defined to be a core cover for a diversity query Q on relation R with d-order δ and limit k if there exists ℓ positive integers $(\beta_1, \cdots, \beta_\ell)$ such that (a) $\sum_{i=1}^{\ell} \beta_i = k$ and (b) for each $t_i \in C$ and for each subset of β_i matching records $S_i \subseteq R$ that is covered by t_i, $\bigcup_{i=1}^{\ell} S_i$ is a diverse result set for Q.*

Thus, each tuple in a core cover C covers at least one tuple in a diverse result set S. We refer to $(\beta_1, \cdots, \beta_\ell)$ as the *core cover assignment for Q*. For the case where $\ell = k$, the core cover assignment for Q is trivially given by $\beta_i = 1$ for each $i \in [1, \ell]$. If $\ell < k$, then there will be duplicate δ-tuples in S and the core cover assignment essentially allocates the distribution of the duplicates among the tuples in C to ensure that S is a diverse result set.

Example 5.1 *Consider a query Q on R with $\delta = (B, SS)$, a single selection predicate "#Core = 4", and a limit of 5. Consider a set of (B,C,SS)-tuples, $C = \{t_1, t_2, t_3, t_4\}$, where $t_1 = (Acer, 4, 13.3)$, $t_2 = (Acer, 4, 14.1)$, $t_3 = (Acer, 4, 17.3)$, and $t_4 = (Lenovo, 4, 13.3)$. Then, C is a core cover for Q with a core cover assignment $(1, 1, 1, 2)$. That is, there exists a diverse result set $S \subseteq R$ for Q where each of the tuples in $\{t_1, t_2, t_3\}$ covers one tuple in S, and t_4 covers two tuples in S. Based on R in Figure 1(a) and the core cover assignment $(1, 1, 1, 2)$, there are two possible diverse result sets for Q corresponding to the two sets of RIDs: $\{10, 12, 13, 17, 18\}$ and $\{11, 12, 13, 17, 18\}$.*

RID	*B*	*SS*
10/11	*Acer*	*13.3*
12	*Acer*	*14.1*
13	*Acer*	*17.3*
17	*Lenovo*	*13.3*
18	*Lenovo*	*13.3*

(a) Diverse result set for Q

RID	*B*	*SS*
10	*Acer*	*13.3*
11	*Acer*	*13.3*
12	*Acer*	*14.1*
13	*Acer*	*17.3*
17/18	*Lenovo*	*13.3*

(b) Non-diverse result set for Q

Note that although there are two tuples in R (with RIDs 10 and 11) covered by t_1, $(2, 1, 1, 1)$ is not a core cover assignment for Q as illustrated by the result sets shown above: the result set in (a) is more balanced than that in (b) wrt Brand attribute. □

The concept of a core cover provides a useful design framework to consider techniques for computing diverse query results. Re-examining `OnePass` [8] with this framework, we see that the core cover C computed by `OnePass`, which is organized using a trie, is characterized by the following two properties: (P1) $|C| = k$, and (P2) all the tuples in C are δ-tuples. As `OnePass` is designed for SDQs, δ is the same as a pre-determined index key α, and `OnePass` uses B$^+$-trees to retrieve δ-tuples to compute C. This is a reasonable approach when δ is the same as α. But as we explained in Section 4, this index design becomes unacceptable when adapted to `OnePass`^{D} for DDQs as using a B$^+$-tree index scan (with key α) to retrieve diverse δ-tuples could be extremely inefficient when α and δ are very different.

To avoid the pitfall of `OnePass`^{D}, we make the observation that since the tuples in a core cover are δ-prefix tuples (of which δ-tuples are just a special case), a better index design is to support the retrieval of δ-prefix tuples (instead of δ tuples). Thus, instead of supporting only a single type of index scan with a single index key α, a more flexible index design is to support multiple types of index scans using α-prefixes as keys to efficiently retrieve α-prefix tuples to form δ-prefix tuples for the core cover.

The rest of this section presents our new index technique to evaluate diversity queries. Our approach consists of two data structures: a novel disk-based *diversity index or D-Index*, which supports efficient index scans with α-prefix

keys (Section 5.2); and a run-time, main-memory structure, called the *result trie*, to organize the tuples in the core cover and guide the index traversal (Section 5.3). We give an overview of how these structures operate together to evaluate diversity queries in Section 5.4, and establish a sufficient condition for a result trie to be a core cover for a query in Section 5.5.

5.2 Diversity Index

A D-Index I on a relation R with index key $\alpha = (A_1, \cdots, A_n)$ is a height-balanced trie-like structure on the set of tuples $\pi_\alpha(R)$. The index consists of $n+1$ levels, $L_0, L_1, \cdots, L_n$, where each L_i corresponds to attribute A_i, $i \in [1, n]$. L_0 consists of a single root node, denoted by N_{root}. Each node N at L_i, $i \in [1, n]$, corresponds to a unique α_i-tuple, denoted by $ptup_\alpha(N)$. Thus, each L_i contains $|\pi_{\alpha_i}(R)|$ nodes, $i \in [1, n]$. A node N at L_i, $i \in [1, n-1]$, is the parent node of another node N' at L_{i+1} if $ptup_\alpha(N)$ is a proper prefix of $ptup_\alpha(N')$.

Each node N at L_i, $i \in [1, n]$, consists of the following information: (1) $ptup_\alpha(N)$, the α-prefix tuple corresponding to N; and (2) the RID, denoted by $rid(N)$, of some tuple in R that is covered by $ptup_\alpha(N)$. $ptup_\alpha(N)$ enables the retrieval of descendant index nodes of N while $rid(N)$ enables the retrieval of a tuple that is covered by $ptup_\alpha(N)$.

Example 5.2 *Figure 1(b) shows the D-Index with index key (Brand, #Core, ScrnSze, BatLife) on R (Figure 1(a)). If N denotes the left child node of the node "Acer", then* $ptup_\alpha(N) = (Acer, 2)$ *and* $rid(N) \in \{7, 8, 9\}$. □

In addition, the root node N_{root} of I also maintains statistics on the number of distinct values for each attribute in α, denoted by $count_{N_{root}}(A_j)$; i.e., for each attribute A_j, $j \in [1, n]$, we have $count_{N_{root}}(A_j) = |\pi_{A_j}(R)|$. These statistics are used for checking certain property of the result trie (to be described in Section 6.3).

A D-Index I can be used to evaluate a diversity query Q if all the d-order attributes α and selection predicate attributes θ occur in the index key α of I.

Definition 5.2 (matching index node) *A node N in a D-Index I is defined to be a matching index node for a diversity query Q if* $ptup_\alpha(N)$ *is a matching tuple for Q.*

The overall idea of using an index I to evaluate Q is to retrieve δ-prefix tuples from the matching index nodes accessed to progressively compute a core cover for Q. Specifically, for each index node N accessed during the traversal of I, if N is matching index node, the α-prefix tuple corresponding to N (i.e., $ptup_\alpha(N)$) is used to update a core cover for Q. However, since the key α of I and the d-order δ of Q are generally different attribute orderings, we need to transform each α-prefix tuple t retrieved from I to its corresponding δ-prefix tuple to update a core cover for Q. We refer to this transformed tuple as the *maximal δ-prefix tuple of t.*

Definition 5.3 (maximal δ-prefix) *Given two attribute orderings of R,* $\alpha_i = (A_1, \cdots, A_i)$ *and and* $\delta = (D_1, \cdots, D_m)$*, we define the maximal* δ*-prefix of* α_i *to be* $(D_1, \cdots, D_j)$*,* $j \in [1, m]$*, if (1) the set of attributes* $\{D_1, \cdots, D_j\}$ *occurs in* α_i *and (2) either* $j = m$ *or* D_{j+1} *does not occur in* α_i*. The maximal* δ*-prefix of* α_i *is defined to be nil if* D_1 *does not occur in* α*.*

Definition 5.4 (maximal δ-prefix tuple) *Given two attribute orderings of R,* $\alpha_i = (A_1, \cdots, A_i)$ *and* $\delta = (D_1, \cdots, D_m)$*, and a* α_i*-tuple t, we define the maximal* δ*-tuple of t to be* $\pi_{\delta_j}(t)$*, where* δ_j *is the maximal* δ*-prefix of* α_i*.*

Given an index node N in I, we use $ptup_\delta^{max}(N)$ to denote the maximal δ-tuple of $ptup_\alpha(N)$.

Example 5.3 *Consider* $\alpha = (A, B, C, D, E)$ *and* $\delta = (C, A, E)$*. The maximal* δ*-prefix of* α_4 *is* (C, A)*. Given a* α*-tuple* $t = (1, 2, 3, 4, 5)$*, the maximal* δ*-tuple of t is* $(3, 1, 5)$*. Consider a query Q with* $\delta = (B, SS, BL)$ *and let N denote the parent node of the rightmost leaf node in the D-Index with* $\alpha = (B, C, SS, BL)$ *in Figure 1(b). Then* $ptup_\delta^{max}(N) = (Lenovo, 13.3)$. □

5.3 Result Trie

To keep track of the maximal δ-prefix tuples that form a core cover for Q, we use a main-memory structure called the *result trie* (denoted by T).

The result trie T consists of at most $m+1$ levels, $L_0, L_1, \cdots, L_m$, where each L_i corresponds to an attribute D_i, $i \in [1, m]$, in the d-order δ of Q. L_0 consists of a single root node, denoted by V_{root}. Each node V at L_i, $i \in [1, m]$, corresponds to a δ_i-tuple, denoted by $ptup_\delta(V)$. A node V at L_i, $i \in [1, m-1]$, is the parent node of another node V' at L_{i+1} in T if $ptup_\delta(V)$ is a proper prefix of $ptup_\delta(V')$.

Each node V of T consists of the following information: (1) $ptup_\delta(V)$, the δ-prefix tuple associated with V; and (2) a set of entries, denoted by $entry(V)$, where each entry $e = (\rho, rid)$ corresponds to an index node N such that $\rho = ptup_\alpha(N)$, $rid = rid(N)$, and $ptup_\delta^{max}(N) = ptup_\delta(V)$. Note that $entry(N_{root}) = \emptyset$.

Definition 5.5 (tree size) *The size of a subtree* T' *of a result trie, denoted by* $size(T')$*, is defined to be the number of leaf nodes in* T'*.*

We use $cover(T)$ to denote the set of δ-prefix tuples corresponding to the leaf nodes of T; i.e., $cover(T) = \{ptup_\delta(V) \mid V$ is a leaf node in $T\}$.

Example 5.4 *Figure 3(h) shows an example result trie wrt a query with* $\delta = (Brand, ScrnSze, BatLife)$*. We have* $cover(T) = \{(Acer, 13.3, 5), (Acer, 14.1, 5), (Acer, 17.3), (Lenovo)\}$*. If V denotes the rightmost child node of the node "Acer", then* $ptup_\delta(V) = (Acer, 17.3)$. □

Note that our result trie differs from the trie used in [8]: our trie is not necessarily height-balanced, and it requires a more intricate maintenance procedure (Section 5.5) as the tuples are inserted into it in a random rather than a sequential order.

5.4 Overview of Query Evaluation

Our overall approach to evaluate a diversity query Q using a D-Index I and result trie T works as follows. For each matching index node N accessed in I, we update T with $ptup_\delta^{max}(N)$. Thus, the result trie is used to organize the retrieved $ptup_\delta^{max}(N)$ tuples, which is in turn used to guide the index traversal to construct a core cover for Q efficiently with a small number of index node accesses.

If the result trie satisfies a sufficient condition for $cover(T)$ to form a core cover for Q (discussed in Section 5.5), the index traversal terminates and $cover(T)$ is used to derive a

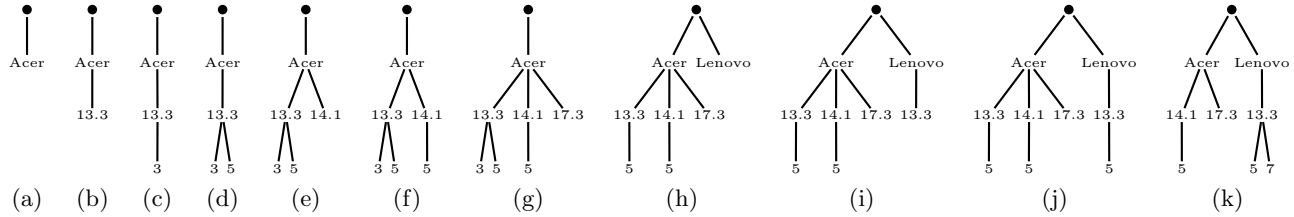


Figure 3: Sequence of updates to result trie by `D-tree` evaluation in Example 6.3

diverse result set for Q as follows. Let $\{V_1, \cdots, V_\ell\}$ denote the set of leaf nodes in T and $(\beta_1, \cdots, \beta_\ell)$ denote the corresponding core cover assignment for Q. Then the *rid* entries from $entry(V_i)$ will be used to retrieve β_i tuples to form the result set for Q. If $|entry(V_i)| < \beta_i$, then we need to retrieve additional matching tuples by using the ρ entries from $entry(V_i)$ to access additional matching index nodes. The core cover assignment is computed to ensure that the trie representation of the derived result set is as balanced as possible so that it is a diverse result set; the details are given elsewhere [5].

5.5 Sufficient Condition for Core Cover

In this section, we establish a sufficient condition for $cover(T)$ to be a core cover for a query Q with limit of k.

Definition 5.6 (diverse trie) *A result trie T for a query Q on relation R with d-order δ is a diverse trie if for any set of matching records $S \subseteq R$, $|S| = |cover(T)|$, that is covered by $cover(T)$, S is a diverse result set of size $|S|$.* □

Definition 5.7 (expandable node) *We say that a node V in a result trie is expandable if it is possible to add a new child node to V. The new child node must correspond to a yet-to-be accessed matching index node.*

Definition 5.8 (balanced node) *A node V in a result trie is defined to be balanced if for each child subtree T_i of V, the difference between $size(T_i)$ and $size(T')$ is at most one, where T' is the largest child subtree (in terms of $size()$) of V; i.e., $size(T') - size(T_i) \leq 1$.*

Definition 5.9 (balanced-diverse (b-diverse) tree) *A subtree T rooted at a node V in a result trie is defined to be a balanced-diverse (or b-diverse) tree if one of the following conditions hold: (1) V is a leaf node, or (2) V is an internal node and either (a) the number of child nodes of V is equal to $size(T)$, or (b) V is balanced and not expandable, and each child subtree of V is a b-diverse tree.*

The following result states that a b-diverse result trie T is a sufficient condition for T to be a diverse result trie.

Lemma 5.1 *If a result trie T is b-diverse, then T is a diverse trie. In addition, if $|cover(T)| = k$, then $cover(T)$ is a core cover for Q.* □

Definition 5.10 (k-sufficient tree) *A subtree T rooted at a node V in a result trie is defined to be a k-sufficient tree if one of the following conditions hold: either (1) V is the root node and $size(T) = k$; or (2) V is not the root node, the subtree rooted at the parent node V_p of V is k-sufficient, and the difference between $size(T)$ and $size(T')$ is at most one, where T' is the largest child subtree (in terms of $size()$) of V_p (i.e., $size(T') - size(T) \leq 1$).*

The following result states that if a subtree T' in a result trie T is a k-sufficient tree, then increasing $size(T')$ will not improve the diversity of T.

Lemma 5.2 *If T is a k-sufficient result trie for a query Q, then there exists a diverse result set S for Q such that for each k-sufficient subtree T' rooted at V in T, the number of tuples in S that are covered by $ptup_\delta(V)$ is at most $size(T')$.*

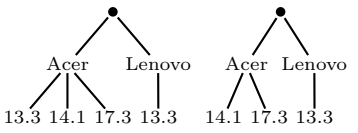
Definition 5.11 (k-optimal tree) *A tree T is k-optimal if T is both b-diverse as well as k-sufficient.*

Example 5.5 *Consider a D-Index I on R with $\alpha = (B, C, SS, BL)$, and a query Q on R with $\delta = (B, SS, BL)$, a single selection predicate "#Core = 4" (i.e., $\theta = \{C\}$), and a limit of 4. Figure 3 shows a sequence of the states of the result trie as it is updated with the δ-prefix tuples corresponding to a specific sequence of accessed index nodes. The node "Acer" in Figure 3(f) is expandable as it is possible to add a new child node "17.3" to it; however the node "Acer" is not expandable in both Figures 3(g) and (k). The root node in Figure 3(h) is not balanced since the size of its left subtree is 3 while that of its right subtres is 1; however, the root node in Figure 3(k) is balanced since the size of each of its child subtrees is 2. In Figure 3(g), the subtree rooted at the node "Acer" is 4-optimal as it is both b-diverse and 4-sufficient; however, the entire trie is 4-sufficient but not b-diverse. In Figure 3(i), the subtree rooted at the node "Acer" is 4-optimal, while the subtree rooted at the node "Lenovo" is b-diverse but not 4-sufficient; the entire trie is 4-sufficient but not b-diverse. Finally, in Figure 3(k), the entire trie is 4-optimal.* □

Based on Lemmas 5.1 and 5.2, we have the following sufficient condition for a result trie to form a core cover for a query.

Theorem 5.1 *If for each node V in a result trie T, the subtree rooted at V is k-optimal or V is not expandable, then there exists a subtree T' of T such that $cover(T') \subseteq cover(T)$ and $cover(T')$ is a core cover for Q. In addition, if T is k-optimal, then $T' = T$.* □

Example 5.6 *Consider a D-Index I on R with $\alpha = (B, C, SS, BL)$, and a query Q on R with $\delta = (B, SS)$, a single selection predicate "#Core = 4", and a limit of 4. In the result trie T shown in Figure 4(a), although T is 4-sufficient, T is not b-diverse and therefore also not 4-optimal. However, observe that Theorem 5.1 applies to T: each node in the subtree rooted at "Acer" is 4-optimal, and the root node as well as each node in the subtree rooted at "Lenovo" is not expandable. Therefore, there exists a subtree T' of T (shown in Figure 4(b)) such that $cover(T')$ is a core cover for Q. Indeed, Figure 4(c) shows a diverse result set for Q that is covered by $cover(T')$.* □



(a) T (b) T'

RID	B	SS
12	Acer	14.1
13	Acer	17.3
17	Lenovo	13.3
18	Lenovo	13.3

(c) S

Figure 4: Example for Theorem 5.1

Note that although Lemma 5.1 provides a sufficient condition for $cover(T)$ to be a core cover for Q, it is not efficient to use this condition alone to guide index navigation to compute the query results as it can lead to many useless index access that retrieve δ-prefix tuples that do not contribute to the final result trie. For efficiency reason, we therefore combine the balanced-diverse and k-sufficient properties in Theorem 5.1 as a stronger sufficient condition for $cover(T)$ to be a core cover for Q. The following example illustrates this requirement.

Example 5.7 *Consider a D-Index I on R with $\alpha = (B, C, SS, BL)$, and a query Q on R with $\delta = (SS, BL)$, a single selection predicate "#Core = 2", and a limit of 4. In the result trie T_1 shown in Figure 5(a), the subtree T' rooted at "14.1" is both b-diverse and 4-sufficient (i.e., 4-optimal). Since T' is 4-sufficient, by Lemma 5.2, it is actually unnecessary to access further index nodes to expand T' since there exists a diverse result set S for Q where the number of records in S covered by (14.1) is no larger than $size(T') = 3$. Indeed, Figure 5(b) shows such a diverse result set for Q. If we had not used this k-sufficient property, then we could have access other unnecessary index nodes (e.g., (Acer,2,14.1,6)) that are covered by (14.1).* □

(a) T

RID	SS	BL
3	14.1	3
4	14.1	5
6	15.4	3
9	15.4	7

(b) S

Figure 5: Example for the k-sufficient property

6. D-INDEX VARIANTS

In this section, we present the key ideas of two instantiations of D-Index: `D-tree` is the simpler variant, which traverses the index in a DFS manner, while `D+-tree` is an improved variant to address the limitations of `D-tree`. The detailed evaluation algorithms for `D-tree` and `D+-tree` are given elsewhere [5]. We use I to denote a D-Index on a relation R with index key $\alpha = (A_1, \cdots, A_n)$.

6.1 Relevant Index Levels (RI-levels)

A D-Index I can be used to evaluate Q if all the attributes in δ and θ occur in α. Note that the ordering of the attributes in δ and θ can be different from α, and α can contain attributes that do not occur in δ or θ.

In general, not all of the index levels in I are relevant and useful for evaluating Q. We classify an index level L_i (corresponding to attribute A_i) as a *relevant index level (or RI-level)* for Q if it satisfies the following four conditions. First, A_i must be relevant for evaluating Q; i.e., A_i must be a diversity attribute in δ or a selection predicate attribute in θ. Second, α_i must contain all the selection predicate attributes in θ. This is necessary to enable checking whether $ptup_\alpha(N)$ for an accessed index node N at L_i is a matching tuple for Q. Third, if A_i corresponds to a diversity attribute D_j in δ, then α_i must contain all the attributes in δ_j. Recall that each matching tuple $ptup_\alpha(N)$ needs to be transformed to its maximal δ-prefix tuple to update the result trie. Therefore, if α_i does not contain some diversity attribute D_r, $r < j$, then it means that the maximal δ-prefix of α_i is at most δ_{r-1}, which implies that the additional values of attributes $\{D_r, D_{r+1}, \cdots, D_j\}$ retrieved from $ptup_\alpha(N)$ are not utilized at all. In this case, we are better off accessing L_{r-1} instead of L_i. Finally, if A_i corresponds to a selection predicate attribute in θ, then α_i must contain the first diversity attribute D_1. Otherwise, the maximal δ-prefix of α_i is empty which means that the index nodes accessed from L_i are useless for updating the result trie.

Example 6.1 *Consider a D-Index I with $\alpha = (A, B, C, D, E, F, G)$ and a query Q with $\delta = (E, C, G, A)$ and $\theta = \{B\}$. Q can be evaluated using I since α contains all the attributes in δ and θ. Only L_5 and L_7 (corrp. to E and G) are RI-levels. L_1 (corrp. to A) violates the second condition, L_2 (corrp. to B) violates the fourth condition, L_3 (corrp. to C) violates the third condition, and L_4 and L_6 (corrp. to D and F) violate the first condition.* □

Trie implementation. As Example 6.1 illustrates, the RI-levels for a query Q are not necessarily consecutive levels in I. Given an index node N, there are two basic access patterns in D-Index: the first is to access the next node after N at the same index level, and the second is to access the first descendant node of N at some RI-level. To efficiently support these access patterns and avoid the overhead of accessing nodes at non-RI levels, we implement each D-Index as a collection of B$^+$-trees. Specifically, for each level L_i in I, the entries in L_i are indexed by a B$^+$-tree with index key α_i; thus, there is one leaf entry in the B$^+$-tree for each level-i index node N in I, and the leaf entry contains its key value $ptup_\alpha(N)$ and $rid(N)$. In this way, the B$^+$-trees corresponding to non-RI levels for Q will not be accessed for evaluating Q.

6.2 Definitions & Notations

Before we present the ideas behind the two index variants, we first introduce several additional definitions and notations.

Definition 6.1 (corresponding T-node of N) *Given a node N in a `D-tree` index, we say that a node V in the result trie T is the corresponding T-node of N if $ptup_\delta(V)$ is $ptup_\delta^{max}(N)$.*

In this paper, we use N to denote an index node in I and use V to denote a node in the result trie T. Given a node V in the result trie I, we use T_V to denote the subtree of the result trie T rooted at V. Given an index node N in I, we use T_N to denote the subtree of the result trie T rooted at the corresponding T-node of N.

Definition 6.2 (heavy/light leaf node) *A leaf node V in T is defined to be a heavy (light) leaf node if for each ancestor node V' of V in T, the subtree rooted at V' is the largest (smallest) subtree (in terms of size()) among its sibling subtrees.*

Example 6.2 *Let N denote the node in the D-Index in Figure 1(b) with $ptup_\alpha(N) = (Acer, 4, 14.1, 5)$. The corresponding T-node of N in Figure 3(g) is the node V with*

$ptup_\delta(V) = (Acer, 14.1, 5)$. *In Figure 3(g), the two leftmost leaf nodes are heavy leaf nodes, while the two rightmost leaf nodes are light leaf nodes.* □

6.3 D-tree Index

In this section, we present the key ideas of evaluating a query Q with a `D-tree` index I. The `D-tree` evaluation algorithm traverses the RI-levels of I in a top-down, depth-first manner. For each matching index node N accessed, we update the result trie with the maximal δ-tuple corresponding to N (i.e., $ptup_\delta^{max}(N)$). If the corresponding T-node of N already exists in T as V, and V is a leaf node in T, then we add an entry corresponding to N into $entry(V)$. On the other hand, if V does not exist in T, we add V into T and update $entry(V)$ as described.

If the update would cause $size(T)$ to exceed k, we first need to select a "victim" tuple from T, denoted by $ptup_\delta(V)$, where V is some leaf node in T, and decide if replacing $ptup_\delta(V)$ by $ptup_\delta^{max}(N)$ would improve the diversity of T. To maximize the diversity of T, we should pick V to be a heavy leaf node. For instance, consider the result trie shown in Figure 3(g) from Example 5.5, where the two leftmost leaf nodes are heavy leaf nodes; clearly, replacing any one of these leaf nodes is better for the diversity of T than replacing any one of the non-heavy leaf nodes.

Having selected a victim tuple $ptup_\delta(V)$, we need to determine whether the replacement would improve the diversity of T. We use a simple sufficient condition to detect whether its diversity would be affected: if V' is the corresponding T-node of N after $ptup_\delta^{max}(N)$ has been inserted into T, V_a is the youngest ancestor node of V with at least two child nodes, and V_a is an ancestor of V', then the replacement does not affect the diversity of T.

Thus, if this sufficient condition holds, we do not update T with $ptup_\delta^{max}(N)$. Continuing with the example trie T_g in Figure 3(g), our approach would not update T_g if $ptup_\delta^{max}(N)$ is say $(Acer, 13.3, 7)$ but we would update T_g if $ptup_\delta^{max}(N)$ is $(Lenovo)$. Thus, $size(T)$ does not decrease as the index evaluation progresses and $size(T)$ is at most k.

For each accessed index node N, we proceed with the DFS-traversal from N to its next descendant node (at the next RI-level) if T_N is not k-optimal. Thus, when the index traversal terminates, Theorem 5.1 guarantees that $cover(T)$ is a core cover for Q. A diverse result set for Q is derived from $cover(T)$ as described in Section 5.4.

Example 6.3 *Consider again Example 5.5. There are three RI-levels corresponding to attributes C, SS, and BL. Figure 3 shows the sequence of updates to the result trie as the* `D-tree` *is traversed to evaluate Q. In each of Figures 3(a) to (f), T is not 4-sufficient. The insertion of (Acer,17.3) in Figure 3(g) causes T to become 4-sufficient, but T is not b-diverse as V_{root} is still expandable. In Figure 3(h), the insertion of (Lenovo) replaces (Acer,13.3,3); and in Figure 3(i), the insertion of (Lenovo,13.3,7) replaces (Acer,13.3,5). At this point, T is 4-optimal as it is both 4-sufficient and b-diverse.* □

To check if a level-i node V in T is expandable, we use the following sufficient condition: if the number of child nodes of V in T is less than the number of distinct values of attribute D_{i+1}, which is obtained from the statistic $count_{N_{root}}(D_{i+1})$ stored in the index's root node, then V is expandable. For the remaining properties (i.e., balanced node, diverse tree, and k-sufficient tree), they can be checked directly based on their definitions or checked more efficiently by incrementally maintaining additional information with each node (e.g., maintaining a flag to indicate whether a node is balanced).

6.4 D$^+$-tree Index

One drawback of `D-tree` is that the DFS-traversal of the index nodes could result in the retrieval of many matching index nodes that do not contribute to the eventual query's core cover; we refer to such index nodes as *useless index nodes*. For instance, in Example 6.3, the three index nodes retrieved to form the result subtrie rooted at $(Acer, 13.3)$ in Figure 3(d) turn out to be useless index nodes as the subtrie was replaced in the final result trie in Figure 3(k).

To reduce the number of useless index node access, we propose an improved variant of `D-tree`, called the `D+-tree`, which differs from `D-tree` in three key ways. First, `D+-tree` traverses the index nodes in a level-wise manner to alleviate the drawback of a DFS-traveral of the index nodes.

Second, `D+-tree` uses additional statistics information to optimize the update of the result trie T so that for each accessed index node N, it is possible to not only add a new node V in T (i.e., V is the T-node corresponding to N) but also know about the number of child nodes of V (but not their contents) in T. We refer to such child nodes as *virtual child nodes (or child vnodes)*. This "look-ahead" capability essentially provides a cost-effective means to construct a larger and more informative result trie (with vnodes) without having to first pay the cost to access the index nodes corresponding to these vnodes. If if turns out that a vnode is subsequently replaced (i.e., its correponding index node is actually useless), we would have saved the index access cost for the replaced vnode.

Third, unlike the `D-tree` where it traverses from one RI-level to the next immediate RI-level, `D+-tree` uses a cost model to determine the next "best" RI-level to access from a given index node. In this way, `D+-tree` is able to further optimize performance by judiciously accessing a selected subset of RI-levels.

Additional statistics. To support the look-ahead capability in `D+-tree`, we extend the statistics information that is stored only with the root node in `D-tree` to every node in `D+-tree`. Specifically, for each level-i node N in a `D+-tree`, we maintain statistics on the number of distinct values for each "descendant" attribute in the index subtree rooted at N, denoted by $count_N(A_j)$; i.e., for each attribute A_j, $j \in [i+1, n]$, we have $count_N(A_j) = |\{t.A_j \mid t \in R, ptup_\alpha(N) \text{ covers } t\}|$. Note that the statistics stored at the index root node are the same for both `D-tree` and `D+-tree`.

Example 6.4 *Let N denote the node labeled "Acer" in the* `D+-tree` *index I in Figure 1(b). We have $count_N(C) = 2$, $count_N(SS) = 4$, and $count_N(BL) = 4$.* □

Level-wise traversal. In `D+-tree`, the top-down traversal of selected RI-levels of the index is carried out in two phases. In the first phase, `D+-tree` selects a starting RI-level (say level ℓ) to traverse (based on a cost model) and scans for matching level-ℓ index nodes. For each accessed index node N, the result trie is updated with $ptup_\delta^{max}(N)$ similar to what is done in `D-tree`. Let V denote the corresponding T-node of N after the update of T. If T_V is not k-optimal, the evaluation algorithm will determine the maximum number of child nodes of V, denoted by MC, for $cover(T)$ to be a

core cover for Q, and insert an appropriate number of child vnodes for V so that the total number of its child nodes in T is MC. Note that vnodes must be leaf nodes in T.

At the completion of the first phase, the result trie T constructed is height-balanced up to level j, where δ_j is the maximal δ-prefix of α_ℓ, with possibly some vnodes at level $j+1$. If T contains vnodes or it is not k-optimal, we begin the second phase of scanning other RI-levels of I which operates by performing a top-down, breadth-first traversal of the result trie starting with level j. Suppose that the algorithm is currently scanning level-i of the result trie, $i \in [j, m)$, and D_i occurs as attribute A_r in α. For each level-i result trie node V accessed, if T_V is not k-optimal or V has child vnodes, then we will start an index scan wrt an index node N. The goal is to retrieve a sufficient number of descendant index nodes of N from I so that their maximal δ-prefix tuples will be inserted into T_V to make V k-optimal (if T_V is not k-optimal), or replace the child vnodes of V (if T_V has child vnodes). To determine N, we pick any one entry (ρ, rid) from $entry(V)$, and let N be the node such that $ptup_\alpha(N) = \rho$. Given N, we use a cost model to select the next "best" RI-level (say ℓ') to access. As before, we update the result trie for each matching level-ℓ' index node N' accessed and if V' is the corresponding T-node of N' and $T_{V'}$ is not k-optimal, we insert an appropriate number of child vnodes for V'.

Since T might have leaf nodes that are vnodes, each update of T should replace a vnode whenever possible. For example, consider the result trie in Figure 6(d) where the two leaf nodes of node $(Lenovo, 13.3)$ are vnodes (indicated by $\circ$ nodes). When T is updated with $(Lenovo, 13.3, 5)$ in Figure 6(e), the update replaces one of the vnodes of $(Lenovo, 13.3)$.

At the completion of the second phase, T does not contain any vnodes, if T is k-optimal, Theorem 5.1 guarantees that $cover(T)$ is a core cover for Q, and the diverse result set is constructed following the same procedure described in Section 5.4. Otherwise, Theorem 5.1 guarantees that there exists a core cover $cover(T')$, $cover(T') \subseteq cover(T)$, and the details of generating a core cover are given elsewhere [5].

Example 6.5 *Consider again Example 5.5 but using* `D+-tree` *as the D-Index. Figure 6 shows the sequence of updates to the result trie as the* `D+-tree` *is traversed to evaluate Q, where the vnodes are indicated by $\circ$ nodes. The first RI-level that* `D+-tree` *chooses to access is the level corresponding to attribute SS; i.e., the RI-level corresponding to attribute B is skipped. Thus, for the evaluation of Q, only two (i.e., corresponding to SS and BL) out of the three RI-levels are accessed. Figure 6(h) shows the result trie at the completion of scanning index nodes at the level for SS. Observe that the* `D+-tree` *evaluation incurs only one useless index node access (i.e., $(Acer, 13.3)$)) compared to three useless index node access using* `D-tree` *in Example 6.3.* □

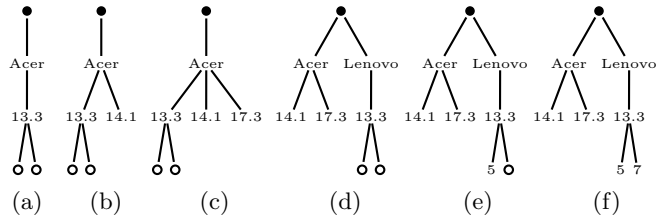


Figure 6: Sequence of updates to result trie by D^+-tree index evaluation in Example 6.5

Besides using the additional statistics to determine the number of child nodes of a result trie node, the additional statistics can also be used to more accurately determine whether a trie node is expandable. Instead of using the approximate statistics in the root node for this purpose (as in `D-tree`), we perform the following for `D+-tree`: whenever we update the result trie with $ptup_\delta^{max}(N)$ to create a new level-j leaf node V in T, we copy the statistic $count_N(D_{j+1})$ from N to V for this purpose, which is more accurate than $count_{N_{root}}(D_{j+1})$.

Cost model for RI-level selection. We now outline how `D+-tree` uses a cost model to select the next "best" RI-level to access wrt a level-ℓ index node N. This RI-level selection problem arises in three cases: (C1) N is the index root node (i.e., selection of the starting RI-level); and (C2) N is the index node corresponding to some trie node V accessed during the breath-first traversal of T, where (a) T_V is not k-optimal or (b) V has some child vnodes. For (C2b), since T_V is already k-optimal, we can simply select the next RI-level below the level of N. For (C1) and (C2a), the procedure is more elaborate as the goal is to pick an RI-level to minimize the overall index access cost to retrieve a target number of index nodes (denoted by num). For (C1), num is equal to the query limit k, while for (C2a), num is equal to maximum possible size of T_V for $cover(T)$ to be a core cover for Q. The maximum subtree size is derived by finding the minimum size of the subtree such that it is k-sufficient; the details of this procedure are given elsewhere [5].

6.5 Implementation Issues

Insufficient RID problem. Note that it is possible that the D-Index might not have sufficient RIDs to answer a query even though there are adequate number of records in the relation R being indexed. This is due to the design of D-Index which stores only a single RID in each index node. To address this problem, one way is to change the design of the last index level (i.e., L_n) so that each level-n index node N now stores the RIDs of all the records in R that are covered by $ptup_\alpha(N)$ instead of just a single RID. With this design, we can retrieve more RIDs associated with a leaf node V in T by first accessing some entry (ρ, rid) from $entry(V)$ and use the α-prefix tuple ρ to retrieve appropriate level-n descendant nodes in I to obtain their RID-lists.

Index key compression. To optimize the performance of the constituent B^+-trees of a D-Index, we compress each index's key values by using a mapping table to map the original attribute values of the keys into compressed forms.

7. PERFORMANCE STUDY

We conducted an experimental study to evaluate the effectiveness of our proposed techniques. Sections 7.1 and 7.2 compare the performance of SDQs and DDQs, respectively, using synthetic datasets. Section 7.3 reports the comparison using real datasets.

Our results show that `D+-tree` has the best performance. For synthetic datasets, `D+-tree` is on average 2× and up to 4.4× faster than `OnePass` for SDQs, and on average 5× and up to 35× faster than `OnePass`^{D} for DDQs. For real datasets, `D+-tree` is on average 1.8× and up to 2.7× faster than `OnePass` for SDQs, and on average 2.2× and up to 3.5× faster than `OnePass`^{D} for DDQs.

Data sets. We generated four synthetic tables, $R_1, \cdots, R_4$, by computing the join of the *lineitem*, *part*, *customer*, and *orders* relations from the TPC-H benchmark using four different scale factors (SF). The properties of these tables are as follows:

Relation	SF	Size (GB)	No. of tuples (million)
R_1	0.75	1.03	4
R_2	4.4	4.83	18.73
R_3	16	9.9	38.35
R_4	36	15	56.35

Each R_i consists of 10 attributes; for convenience, we use $A, \cdots, J$, respectively, to denote the attributes *linenumber*, *discount*, *tax*, *returnflag*, *container*, *shipinstruct*, *shipmode*, *linestatus*, *nationkey*, and *orderstatus*.

The synthetic datasets are evaluated using the following 5 SDQs (Q_1 to Q_5) and 5 DDQs (Q_6 to Q_{10}):

Query	θ
Q_1	A
Q_2	C
Q_3	F
Q_4	C,F
Q_5	A,C,F

Query	θ	Diversity Ordering, δ
Q_6	A	A,F,B,C,D,E,J,G,H,I
Q_7	A	B,C,D
Q_8	A	B,D,C
Q_9	A	C,D,B
Q_{10}	A	D,B,C

All the SDQs share the same d-order $\delta = (A, B, C, D, E, F, G, H, I, J)$. Recall that θ represents a query's set of selection predicate attributes (SPA). To be fair to `OnePass` [8], we used only equality selection predicates for all queries.

Algorithms. We compared our proposed `D-tree` and `D`$^+$`-tree` against `OnePass` [8] and `OnePass`^{D}. Recall from Section 4 that `OnePass`^{D} is an extended variant of `OnePass` to evaluate DDQs; we incorporated D-Index's result trie structure into `OnePass`^{D} to support the random order of trie updates. Since `Probe` performed similarly to `OnePass` for SDQs [8] and is expected to be worse than `OnePass`^{D} for DDQs (Section 4), we omit the comparison against `Probe` and its extension. We also evaluated the performance of two sequential scan techniques: *TableScan* scans the relation while *DIndexScan* scans the last RI-level of a D-Index. However, as these two techniques performed significantly worse than `D`$^+$`-tree` (`D`$^+$`-tree` is about 50× and 100× faster than *DIndexScan* and *TableScan*, respectively), we omit these two techniques in this paper.

All the algorithms were implemented in PostgreSQL 9.0.2: we extended PostgreSQL's GIN index to support the skip operations for `OnePass` [8] and `OnePass`^{D}, and both `D-tree` and `D`$^+$`-tree` were implemented as a collection of B$^+$-trees (Section 6.1).

For each table R_i, we built a `D-tree` and `D`$^+$`-tree` with index key $\alpha = (A, \cdots, J)$, and built the B$^+$-trees of `OnePass` and `OnePass`^{D} with α as the index key. Our implementation shows that `D`$^+$`-tree` index is about 4 times smaller than the GIN index used in `OnePass` and `OnePass`^{D}: As an example, for the 15GB table, the size of the `D`$^+$`-tree` is only 1.9GB while the size of the GIN index is 8.5GB.

Parameters. We varied the following four experimental parameters: (1) the size of dataset with the default size of 10GB using R_3, (2) the query limit k with a default value of 10, (3) the number of selection predicate attributes (SPA) with a default value of 1, and (4) the position of a SPA with a default value of 1.

For comparing DDQs, we also varied two additional parameters: (1) length of query d-order (i.e., $|\delta|$), and (2) the ordering of the attributes for a given set of diversity attributes.

The experiments were conducted on a PC with a Qual-Core Intel Xeon 2.66Ghz processor, 8GB of memory, one 500G SATA disk and another 750GB SATA disk, running Ubuntu 10.04.4. Both the operating system and PostgreSQL were built on the 500GB disk, while the database was stored on the 750GB disk.

In our experiments, each execution time reported refers to the total running time for a query. Each running time is measured with the query running alone in the database system, and the database system is restarted between queries. Each query is run 5 times, and the reported running time is the average of 3 values excluding the minimum and maximum values.

7.1 Static Diversity Queries

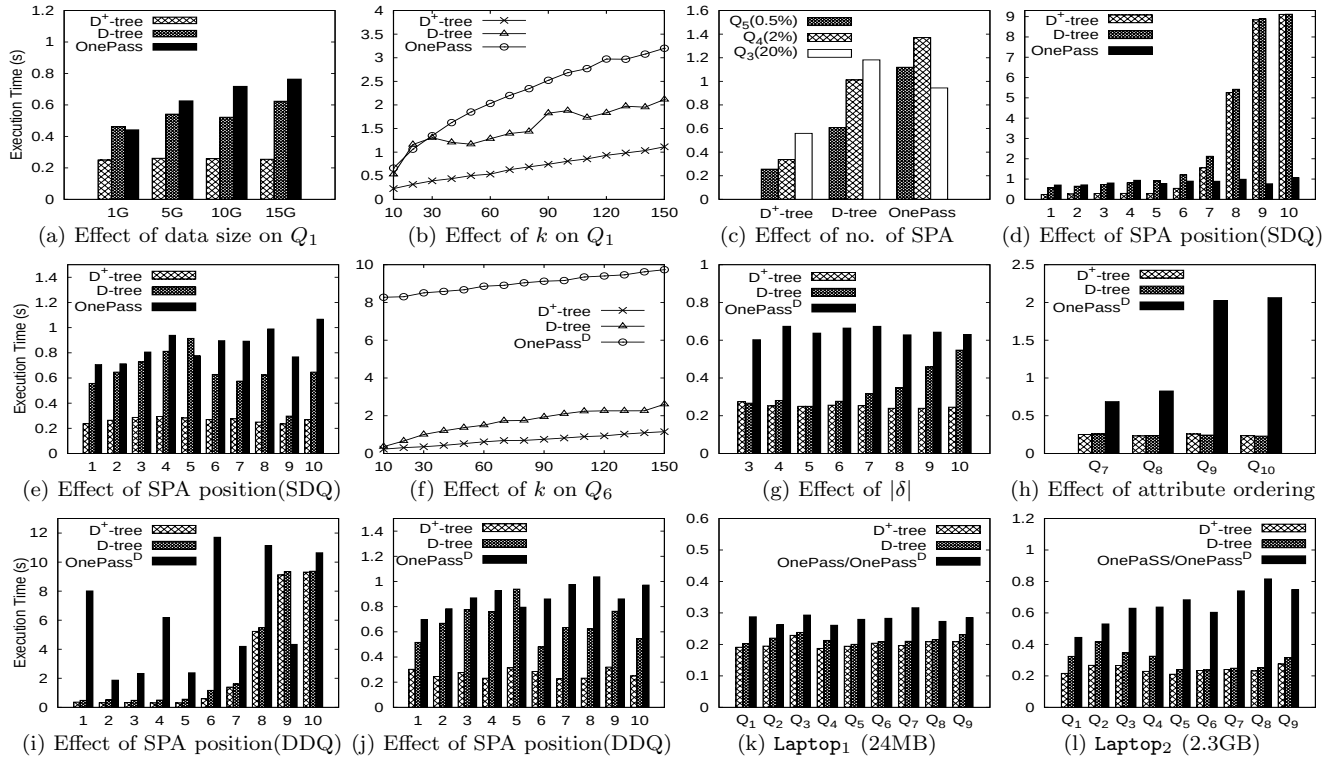
Effect of data size. Fig. 7(a) compares the performance for different data sizes on Q_1. The results show that `D`$^+$`-tree` gives the best performance and it outperforms `OnePass` by an increasing factor of 1.7, 2.4, 2.7, and 3.0 as the data size increases. Observe that while `D`$^+$`-tree` performs similarly for the different data sizes, `OnePass`'s performance worsens with increasing data size. The results demonstrate that `D`$^+$`-tree`'s level-wise index traversal is more effective and scalable than the depth-first traversal of `D-tree`. The results also show that `D-tree` generally outperforms `OnePass`: the reason is that while it is possible for `D-tree` is to terminate its DFS-traversal at any index level, `OnePass` can only terminate its scan at the leaf level. The results for the other SDQs show similar performance trends and are reported elsewhere [5].

Effect of query limit, k. Fig. 7(b) compares the performance for different values of the query limit k on Q_1. Here again, the results show that `D`$^+$`-tree` gives the best performance which outperforms `OnePass` by up to a factor of 3. The number of index entries accessed by `OnePass` increases from 211 to 17723 as k increases from 10 to 150, while that for `D`$^+$`-tree` only increases from 11 to 297. Note that the performance fluctuations for `D-tree` is due to the fact that as k increases, although the number of accessed pages increases, the I/O access pattern also becomes more sequential. The results for other SDQs, available in [5], show similar performance trends.

Effect of number of SPA. Fig. 7(c) compares the performance as the number of selection predicate attributes is varied. We used queries Q_3, Q_4 and Q_5, which have 1, 2, and 3, SPAs, respectively, and query selectivity factors (denoted by *sel*) of 20%, 2%, and 0.5%, respectively.

The results show that `D`$^+$`-tree` gives the best performance and it outperforms `OnePass` by an increasing factor of 1.7, 4.1, and 4.4, as *sel* decreases. For both `D-tree` and `D`$^+$`-tree`, their performance improves (as expected) when *sel* decreases. However, `OnePass` actually performs worse when *sel* drops from 20% to 2%, and then improves when *sel* drops further to 0.5%. There are two factors affecting the performance of `OnePass` when there are multiple SPAs: one is the increase in number and cost of index scans with more SPAs, and the other is the more effective index skips with more SPAs. Thus, `OnePass` performs worse for Q_4 compared to Q_3 as the first factor dominates the second factor; however, it performs better for Q_5 compared to Q_4 as the second factor dominates the first factor.

Effect of SPA position. Fig. 7(d) compares the performance of 10 SDQs with the same d-order of δ and a single SPA whose position varies from 1 to 10. The results show that `OnePass` performs similarly for all of the 10 queries



(a) Effect of data size on Q_1 (b) Effect of k on Q_1 (c) Effect of no. of SPA (d) Effect of SPA position(SDQ)

(e) Effect of SPA position(SDQ) (f) Effect of k on Q_6 (g) Effect of $|\delta|$ (h) Effect of attribute ordering

(i) Effect of SPA position(DDQ) (j) Effect of SPA position(DDQ) (k) $\texttt{Laptop}_1$ (24MB) (l) $\texttt{Laptop}_2$ (2.3GB)

Figure 7: Comparison with synthetic datasets and real datasets

as it is insensitive to the SPA position. In contrast, while both `D-tree` and `D+-tree` perform similarly for the first six queries (i.e., with SPA position between 1 and 6) their performance deteriorate significantly for the last three queries (i.e., when the SPA position is at least 8). The reason is that the size of the first RI-levels for the last three queries are very large.

However, this performance issue with using a single D-Index to evaluate a set of workload queries can be addressed by selecting a set of indexes (wrt to some space constraint) to evaluate the workload. Indeed, we have developed an efficient heuristic for this index selection problem [5], and for this workload of ten queries, it turns out that building an additional D-Index with index key $\alpha' = (A, F, B, H, J, G, I, C, D, E)$ is sufficient to address the performance issue. The total size of the two D-Indexes is only 36% of the size of the single index used by `OnePass`. Fig. 7(e) shows the performance comparison with both `D-tree` and `D+-tree` using this two-index configuration (i.e., each query is evaluated using the more efficient index between the two). The results show that `D+-tree` is consistently the most efficient method. Note that since all the static queries have the same d-order δ, the index key used in the single `OnePass` index (which is equal to δ) is already the optimal index key for evaluating each of the static queries. Therefore, unlike the D-Index, the performance of `OnePass` will remain the same even if additional indexes are created for the `OnePass` approach.

7.2 Dynamic Diversity Queries

Effect of query limit, k. Fig. 7(f) compares the performance for different values of the query limit k on Q_6. The results show that `D+-tree` outperforms $\texttt{OnePass}^D$ by up to a factor of 35. Comparing Fig. 7(f) for DDQs with Fig. 7(b) for SDQs, we observe that the performance of both `D+-tree` and `D-tree` do not vary too much, but the performance of $\texttt{OnePass}^D$ for DDQs is worse than that of `OnePass` for SDQs. This demonstrates that it is not effective to extend `OnePass`, which was designed for SDQs, to handle DDQs. For example, when $k = 10$, $\texttt{OnePass}^D$ scans a total of 1761346 index entries of which only 61 of them are used to update the result trie. This result concurs with our explanation of $\texttt{OnePass}^D$'s expected poor performance in Section 4. The performance results for other DDQs show similar trends and are reported elsewhere [5].

Effect of length of query d-order, $|\delta|$. In this experiment, we examine the effect of varying the length of the query d-order. We generated 8 DDQs, Q_1^3, $\cdots$, Q_1^{10}, from Q_1, where each of these queries is the same as Q_1 except that the d-order of Q_1^i is the length-i prefix of that of Q_1; thus, Q_1^{10} is the same as Q_1.

The results in Fig. 7(g) show that `D+-tree` consistently outperforms $\texttt{OnePass}^D$ by up to a factor of 2.2. Observe that the performance of `D+-tree` is very similar for all the queries; indeed, `D+-tree` selects the same initial RI-level of 3 for all the queries. The performance of $\texttt{OnePass}^D$ is also not too sensitive to $|\delta|$ as it does not seriously affect the number of index pages accessed. For `D-tree`, its performance becomes worse for the last four queries due to an increase in the number of index node access: the number of index pages accessed by `D-tree` for the 8 queries are 3, 4, 8, 10, 20, 27, 37, and 47, respectively.

Effect of ordering of diversity attributes. In this experiment, we examine the effect of different orderings of a same set of diversity attributes. Fig. 7(h) compares the performance for the queries Q_7, Q_8, Q_9, and Q_{10} which all share the same set of diversity attributes $\{B, C, D\}$. In the following discussion, we use δ_{Qi} to denote the d-order for Q_i.

The results show that the performance of both D-tree and D$^+$-tree are not sensitive to the attribute ordering. This is because the number of RI-levels for these four queries are small: they are 3, 2, 2 and 1 levels, respectively. More importantly, the sizes of these RI-levels are also small. In contrast, the performance of OnePass^D varies rather widely. OnePass^D performs the best for Q_7 with $\delta_{Q7} = (B, C, D)$ because together with the selection attribute A, (A, B, C, D) forms a proper prefix of the index ordering α which enables OnePass^D to perform efficiently. For Q_8 with $\delta_{Q8} = (B, D, C)$, the performance of OnePass^D is slightly worse relative to that for Q_7 because δ_{Q8} with selection attribute A now forms a shorter proper prefix (A, B) of α and its evaluation now requires more skip operations compared to that for Q_7. However, for queries Q_9 and Q_{10}, the performance of OnePass^D becomes significantly worse because both δ_{Q9} as well as δ_{Q10} are ordered drastically differently from α which is not conducive at all for the performance of OnePass^D as explained in Section 4. Thus, OnePass^D performs equally poorly for the last two queries.

Effect of SPA position. Fig. 7(i) compares the performance of 10 DDQs with the same d-order as that of Q_6 and a single SPA whose position varies from 1 to 10. Comparing the performance for DDQs in Fig. 7(i) with that for SDQs in Fig. 7(d), we have two key observations. First, the performance behaviour of D-Index (i.e., D-tree and D$^+$-tree) is similar for both SDQs and DDQs; and OnePass^D outperforms D-Index when the SPA position is 9. Second, while OnePass performs efficiently for for all the SDQs in Fig. 7(d), OnePass^D performs poorly for DDQs in Fig. 7(i). Note that the performance of D-Index depends very much on the size of the starting RI-levels while that of OnePass^D depends on the size of the selected inverted lists. Thus, if the size of the starting RI-levels is much larger than that of the inverted lists, OnePass^D could outperform D-Index.

However, similar to our discussion for SDQs in Fig. 7(e), the performance for evaluating a set of queries could be improved by using more than one index. In Fig. 7(j), we compare the performance of the methods using a set of two indexes. For OnePass^D, the optimal index has key (A,F,B,C,D,E,J,G,H,I), while for both D-tree and D$^+$-tree, the optimal set of two indexes have keys (A,F,B,C,D,E,J,G,H,I) and (A,J,F,G,H,B,I,C,D,E). Comparing the results in Figs. 7(i) and (j), it is clear that the performance of each of the methods improve with an additional index, and D$^+$-tree significantly outperforms OnePass^D in Fig. 7(j). Note that the total size of the two D-Indexes is only 26% of the size of the single index used by OnePass^D.

7.3 Comparison on Real Data Sets

In this section, we present performance results using a real dataset on laptop products extracted from eBay. The original dataset (denoted by Laptop$_1$) is a relation with 11 attributes containing 39,411 laptop records (24MB). We created a larger dataset (denoted by Laptop$_2$) from Laptop$_1$ by duplicating it 100 times. For each of these two datasets, we created four indexes, OnePass, OnePass^D, D-tree, and D$^+$-tree, all with the same index key (B, T, C, M, D, S, P, O), where B, T, C, M, D, S, P, and O denote attributes *brand*, *type*, *condition*, *memory*, *disk*, *screen size*, *processor type* and *operating system*, respectively. We used the following nine diversity queries for this experiment: queries Q_1 to Q_4 are SDQs, while queries Q_5 to Q_9 are DDQs.

The performance results in Figs. 7(k) and (l) shows that the performance gain of D$^+$-tree over OnePass and OnePass^D increase with the data size. For the Laptop$_1$ dataset, Fig. 7(k) shows that D$^+$-tree outperforms OnePass by up to a factor of 1.5 for SDQs and outperforms OnePass^D by up to a factor of 1.6 for DDQs. For the Laptop$_2$ dataset, Fig. 7(l) shows that D$^+$-tree outperforms OnePass by up to a factor of 2.7 for SDQs and outperforms OnePass^D by up to a factor of 3.5 for DDQs.

Q	Selection Predicates	Diversity Ordering, δ	k
Q_1	B = 'HP'	B, T, C, M, D, S, P, O	10
Q_2	B = 'HP'	B, T, C, M, D, S, P, O	20
Q_3	C = 'New'	B, T, C, M, D, S, P, O	10
Q_4	B = 'HP' and C = 'New'	B, T, C, M, D, S, P, O	10
Q_5	B = 'HP'	T, M, C, S, D, P	10
Q_6	B = 'HP'	T, M, C, S	10
Q_7	B = 'HP'	M, D, S, C, T, P	10
Q_8	B = 'HP' and C = 'New'	T, M, S, D, P	10
Q_9	B = 'HP'	T, M, C, S, D, P	20

8. CONCLUSION

In this paper, we have examined the problem of computing diverse query results. We have proposed a novel indexing technique, D-Index, that is based on the concept of computing a core cover, for evaluating both static as well as dynamic diversity queries. We also have designed two instantiations of the D-Index, D-tree and D$^+$-tree. Our comprehensive performance study comparing against the state-of-the-art technique for static diversity queries, OnePass, and its extended variant for dynamic diversity queries, showed that D$^+$-tree outperforms existing techniques on average by a factor of 2.

Acknowledgements We would like to thank the reviewers for their constructive comments and Sharad Mehrotra for his feedback on an earlier draft of the paper. This research is supported in part by NUS Grant R-252-000-453-112.

9. REFERENCES

[1] Result diversity. *IEEE Data Eng. Bull.*, 32(4), 2009.
[2] A. Angel and N. Koudas. Efficient diversity-aware search. In *SIGMOD*, 2011.
[3] G. Capannini, F. M. Nardini, R. Perego, and F. Silvestri. Efficient diversification of web search results. *PVLDB*, 2011.
[4] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Rec.*, 39(1), 2010.
[5] L. Li and C.-Y. Chan. Efficient indexing for diverse query results. Technical report, National University of Singapore, February 2013. http://www.comp.nus.edu.sg/~lilu0355/techreport-DIndex.pdf.
[6] F. Radlinski and S. T. Dumais. Improving personalized web search using result diversification. In *SIGIR*, 2006.
[7] A. D. Sarma, S. Gollapudi, and S. Ieong. Bypass rates: reducing query abandonment using negative inferences. In *KDD*, 2008.
[8] E. Vee, U. Srivastava, J. Shanmugasundaram, P. Bhat, and S. Amer-Yahia. Efficient computation of diverse query results. In *ICDE*, 2008.
[9] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina-Jr., and V. J. Tsotras. On query result diversification. In *ICDE*, 2011.
[10] C. Zhai and J. Lafferty. A risk minimization framework for information retrieval. *Inf. Process. Manage.*, 2006.
[11] C. N. Ziegler, S. Mcnee, J. Konstan, and G. Lausen. Improving Recommendation Lists Through Topic Diversification. In *WWW*, 2005.