# A Novel Index Structure for Multi-key Search

Dongyu Wei[1], Xin Pan[1], Chuan Shi[1,⋆], and Yueguo Chen[2]

[1] Beijing University of Posts and Telecommunications, Beijing, China 100876
[2] Renmin University, Beijing, China

**Abstract.** The linear storage model is widely used to support in-memory multi-key search running on small devices of limited computing capacity, simply because it avoids the maintenance of space-costly and energy-costly indexing structures. However, it only supports sequential multi-key scan which is slow and energy-consuming. We design an index structure called D-Tree to address the problem.

**Keywords:** Storage model, Multi-key search, Space-sensitive, Energy-sensitive.

## 1 Introduction

Multi-key search has been an important function in database systems running on small devices [3]. It is to search tuples within a table constrained by two or more keys. Modern database systems use various indexing techniques (e.g., B+-tree [2], kd-Tree [1] and Bitmap [4]) to support efficient query processing of multi-key search queries. However, for applications running on small devices which have critical physical constraints (e.g., space and energy), the advanced indexes are often not applicable due to their excessive cost in space consumption. Quite often, linear list is more favored by many database systems designed for small devices with physical limitations. Although linear storage model has no extra space overhead and very few maintenance cost, the sequential scan process is however slow and energy-consuming due to the vacancy of indexing supports. In this paper, we address the multi-key search problem in space-limited memory and propose a novel lightweight index named D-Tree for multi-feature datasets.

## 2 The D-Tree Index Structure

### 2.1 Construction Algorithm of D-Tree

Inspired by the dominance tree [5], we propose the D-Tree index structure to support such a multi-key search query with small extra space cost but efficient search speed. A dominance tree [5] is a binary tree, where the left-link field links to its left subtree whose root node is dominated by that node, and the right-link filed links to its right sub-tree whose root node is non-dominated by that node. However,different from dominance tree, the D-Tree has just the child field which point to the dominated nodes, whose data structure can be defined as follows.

⋆ Corresponding author.

```
typedef struct DT{
    int id;  //coordinate in the sibling list
    struct DT *child; //point to the dominated nodes
 }
```

**Algorithm 1.** addinTree(DT *pNode, DT *newNode)

Insert a new node (newNode) into pNode's left subtree when newNode is dominated by pNode.

```
1. if pNode has child then
2.    addinList(pNode, pNode.child, newNode);
3. else
4.    pNode.child = newNode;
5. end if
```

**Algorithm 2.** addinList(DT *pNode, DT *cNode, DT *newNode)

When newNode is inserted into pNode's left subtree, newNode is compared with cNode, a node in the pNode's left child list.

```
while TRUE do
    if cNode is nondominated with newNode then
        if cNode is the last one in the list then
            append newNode to the list;
            return;
        else
            cNode = next node in the list;
        end if
    else if cNode dominates newNode then
        addinTree(cNode, newNode);
        return;
    else if cNode is dominated by newNode then
        remove cNode from the list;
        addinTree(newNode, cNode);
        if cNode is the last node in the list then
            append newNode to the list;
            return;
        else
            cNode = next node in the list;
        end if
    end if
end while
```

**Algorithm 3.** Equality Search

```
Require: query;
    SN = CreateSN(query);
    Create an empty stack; CN = root;
    while CN != NULL do
        if Better(SN,CN) == 1 then
            prune the left branch of CN; CN=CN's right neighbor;
        else if Better(SN,CN) == -1 then
            push CN's left child to stack; CN=CN's right neighbor;
        else if Better(SN,CN) == 0 then
            if SN match CN then
                find a target tuple;
                push CN's left child to stack; CN=CN's right neighbor;
            else
                prune the left branch of CN; CN=CN's right neighbor;
            end if
        end if
        if CN == NULL then
            if stack == NULL then
                return;
            else
                CN = Pop(stack);
            end if
        end if
    end while
```

To construct a D-Tree, nodes are inserted one by one. When a node (called *newNode*) is compared with an existing node (called *cNode*), there are three possible results. 1) *newNode* dominates the *cNode*. *cNode* is removed and inserted into the *newNode*'s left branch. *newNode* will be further compared with original *cNode*'s right neighbors. 2) *newNode* is dominated by *cNode*. The *newNode* will be inserted into the *cNode*'s left branch for further comparison. 3) *newNode* is non-dominated with *cNode*. *newNode* will be further compared with the *cNode*'s right neighbors. The node insertion algorithm of D-Tree is shown in Algorithm 1 and 2.

**Right-link Sort**. In many cases, a key (called hot key) is usually visit many times in queries.We propose the right-link sort method to boost the performance of queries containing a hot key. We will show that such a variation can improve search performance in experiments.

**Equality Search** Here we consider the simplest operation (i.e., "=") for the multi-key search algorithm.The algorithm is shown in Algorithm 3. The search process can be roughly separated into three steps. 1) Input an array list (a sibling chain). 2) Iteratively compare SN with CN through the list. 3) Determine whether go back to 1) or not.

## 3 Experiments

We test the equality search performances on synthetic datasets. Three structures, FS, D-Tree and srD-Tree, are included in the experiments. FS means the multi-key sequential scan on linear list. D-Tree means the multi-key search on D-Tree without right-link sort adjustment. srD-Tree means the multi-key search on D-Tree with right-link sort. One search key is always selected as hot key.

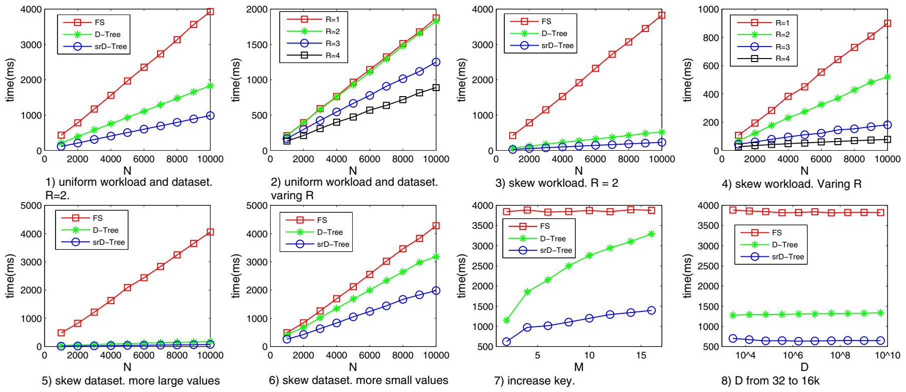

**Fig. 1.** Time efficiency experiments on equality search.$N$ represents the number of tuples, $M$ represents the number of keys indexed by D-Tree, $R$ represents the the number of keys specified in Search Node, $D$ represents the value domain of each key

In the results of Fig. 1(1), the data of a table with four keys are indexed. Each key has a domain of [1,50]. All queries contain 2 search key (i.e., $R = 2$). The results show that D-Tree consistently outperforms the solution of the sequential multi-key scan. srD-Tree is the fastest one. In experiments of Fig. 1(2), we vary the number of search keys from 1 to 4. As shown in the figure, the more search keys used, the faster D-Tree and srD-Tree are. This is because when more values are filled in the search node (SN), SN has larger possibility to be non-dominated with CN.

Different from Fig. 1(1) and (2) which use uniform workload, Fig. 1(3) and (4) generate skew workload using Zipfian distribution[1] with parameter 1.0. Smaller values have larger possibility to be searched. Similar to the experimental setting in Fig. 1(1) and (2), we use 2 search keys in Fig. 1(3) and vary the number of keys in Fig. 1(4). As we expected, D-Tree and srD-Tree further improve the performance. srD-Tree in Fig. 1(4) is over 2 orders of magnitude faster than the sequential scan. The reason is that, when the values in the query become smaller, the target tuples stay closer to the root.

Instead of generating skew workload, we generate skew dataset using Zipfian distribution with parameter 1.0 in Fig. 1(5) and Fig. 1(6). Besides, 3 search keys

[1] `http://en.wikipedia.org/wiki/Zipf's_law`

are used in queries. In Fig. 4(5), large values has large possibility to be generated, while in Fig. 1(6) small values have large possibility of being generated. srD-Tree is nearly two orders of magnitude faster than FS.

In experiments of Fig. 1(7), the impact of $M$ is tested. As shown in Fig. 1(7), when more keys are indexed, the performance of D-Tree and srD-Tree degrade. The reason is that when more keys are indexed, nodes have large possibility to be non-dominated with each other. And the sibling chain, which cannot be pruned by D-Tree, gets longer. Even though D-Tree degrades, it still outperforms FS when 15 keys are indexed.

We also test the impact of value domain in Fig. 1(8). The value domain of all keys is enlarged together. As the value domain is enlarged, fewer duplicate values occur. However, we do not observe the significant change of the performance when varying the value domain of all keys from 32 to 16k. It means that the time efficiency of D-Tree is not sensitive to domain size.

## 4 Conclusions

Data management engines on small devices are faced with common special physical restrictions and limited energy support. They are not quite time-sensitive but have high requirements on space and energy. We propose the novel D-Tree, an in-memory lightweight storage model for multi-key search. The D-Tree can effectively store the dominance relationships of tuples with small extra index space. We design efficient search algorithms based on D-Tree for multi-key search. Extensive experiments show that D-Tree can achieve 2 orders of magnitude improvement than linear scan with very small extra space cost. It indicates that D-Tree can be an effective substitution of linear list on the extreme space limit scenarios, such as smart card and sensor database.

**Acknowledgement.** This work is supported by the National Basic Research Program of China (2013CB329603). It is also supported by the National Natural Science Foundation of China (No. 61375058, 60905025, 61074128) and Ministry of Education of China and China Mobile Research Fund (MCM20123021).

## References

1. Bentley, J.L.: Multidimentional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
2. Comer, D.: The ubiquitous b-tree. ACM Computing Surveys 11(2), 121–137 (1979)
3. Li, X., Kim, Y.J., Govindan, R., Hong, W.: Multi-dimensional range queries in sensor networks. In: SenSys, pp. 63–75 (2003)
4. O'Neil, P., Quass, D.: Improved query performance with variant indexes. In: SIGMOD, pp. 38–49 (1997)
5. Shi, C., Yan, Z., Lu, K., Shi, Z., Wang, B.: A dominance tree and its application in evolutionary multi-objective optimization. In: Information Sciences, pp. 3540–3560 (2009)