An Efficient Indexing Structure for Multidimensional Categorical Range Aggregation Query

Jian Yang[1*], Chongchong Zhao[1], Chao Li[2], and Chunxiao Xing[2]
[1] School of Computer and Communication Engineering, University of Science and Technology Beijing
Beijing, 100083 - China
[2] Research Institute of Information Technology, Tsinghua University
Beijing, 100084 - China
[e-mail: yjian180@gmail.com]
*Corresponding author: Jian Yang

Received February 15, 2018; revised April 26, 2018; revised July 14, 2018; accepted September 1, 2018; published February 28, 2019

Abstract

Categorical range aggregation, which is conceptually equivalent to running a range aggregation query separately on multiple datasets, returns the query result on each dataset. The challenge is when the number of dataset is as large as hundreds or thousands, it takes a lot of computation time and I/O. In previous work, only a single dimension of the range restriction has been solved, and in practice, more applications are being used to calculate multiple range restriction statistics. We proposed MCRI-Tree, an index structure designed to solve multi-dimensional categorical range aggregation queries, which can utilize main memory to maximize the efficiency of CRA queries. Specifically, the MCRI-Tree answers any query in $O(nk^{n-1})$ I/Os (where n is the number of dimensions, and k denotes the maximum number of pages covered in one dimension among all the n dimensions during a query). The practical efficiency of our technique is demonstrated with extensive experiments.

Keywords: Categorical, Multidimensional Indexing, Query, MCRI-Tree

1. Introduction

Categorical range aggregation (CRA) is a type of query which can be regarded as the simultaneous execution of a range aggregation query on multiple datasets, returning a result for each dataset. This requires a lot of computation time and I/O. Previous work only solved CRA with single range restriction. However, it is common to calculate statistical data with multiple range restrictions in modern business management. Let us consider the following query about empolyees from **Table 1**.

Table 1. Data samples about empolyee

ID	Name	Age	Sex	Status	Entry_time	Salary	Position	City
121	Brandon	23	M	single	2015/4	50k	staff	BJ
129	Daphne	27	F	single	2016/1	50k	staff	MH
76	Jeremy	36	M	married	2010/7	95k	Engineer	MH
109	Emily	22	F	single	2013/11	65k	staff	NYC
56	Armand	38	F	single	2011/6	90k	Manager	BJ
43	Leona	46	M	married	2010/8	120k	Manager	NYC
87	Renata	34	F	married	2013/5	70k	Engineer	BJ
132	Norman	28	M	married	2016/2	55k	staff	MH

Now, we give a query:

Q_1: *Find the number of employees whose entry_time is after Jun.2013 in each of the city.* It can be expressed in SQL:

```
SELECT COUNT(*) FROM Table 1
WHERE Entry_time > '2013/6'
GROUP BY City
```

This is a Categorical Range Aggregation Query(CRA). Now, we formally define the problem.

Definition 1 *Categorical Range Aggregation Query(CRA): Given a dataset denoted by D, a query range r, and an aggregation function f. The result of CRA is defined as*:

$$result = \{_{category\_column}G_{f(aggregation\_column)}(\sigma_{range\_column \in r}(D))\} \quad (1)$$

In the above query, the restriction has only one column, however, in a real system, the user does not satisfy the query under a single column restriction, and usually involves multiple columns. Let us continue to consider the following query about **Table 1**.

Q_2: *Find the number of employees whose entry_time is after June.2013, the salary is higher than 70k and the age less than 35 in each of the city.*

It can be expressed in SQL:

```
SELECT COUNT(*) FROM Table 1
WHERE Entry_time > '2013/6' AND Salary > '70k' AND Age < '35'
```

GROUP BY City

Compared with the previous queries on categorical range aggregation, Q_2 has a key difference: This query needs to satisfy multiple range restrictions, not just one. We increase the number of constraints and make queries more flexible and useful. Meanwhile, calculate statistics according to the category of each data item.

In this paper, we will define this query as Multidimensional Categorical Range Aggregation Query, which can be formally defined as follows:

Definition 2 *Multidimensional Categorical Range Aggregation Query (MCRA): Given a relation D, a multidimensional query range $r_1, r_2,...,r_n$, and an aggregation function f, the result of MCRA query is defined as*:

$$result = \{ {}_{category\_column} G_{f(aggregation\_column)} (\sigma_{RC_1 \in r_1 \wedge \cdots \wedge RC_n \in r_n} (D))\} \quad (2)$$

It can be seen that CRA is a special case of MCRA in the case that the query range is single-dimension. In CRA and MCRA problems, the columns can be divided into three types:

- Category column: The query results are grouped by these columns, which has a known limited range of values. For instance, the city in **Table 1** is a category column. In [1], this column is also called a color column.
- Range column: The result of the aggregation comes from the projections of the values of these columns, in other words, the columns will be indexed to speed up the query. Such as the age is the range column in **Table 1**.
- Aggregation column: The values will be aggregated with a specific aggregation function. The aggregation functions are usually SUM, COUNT, AVERAGE, MAX, MIN (and so on).

When create an index structure to solve the MCRA problems, we need to assign each field separately to one of these three columns. However, there is more than one type for each column in the table, which can be multiple. For example, the salary column can be either a range column or a category column, and so on.

1.2 Application and Motivation

Multidimensional Categorical Range Aggregation is motivated by the fact that, in applications where data is naturally divided into categories, a user can limit the attributes in multiple ranges based on their interests.

Queries like Q_2 report statistics about employees in the business and are important for several reasons. First, the statistics provide valuable information for a company, especially for human resource management(HRM). For instance, HRM can develop a flexible pay and benefit framework based on the differences in regional development. Second, the statistics play an important role in HRM, especially for large multinationals in terms of staffing structure in different regions, as it allows the HR to conduct talented people in multiple dimensions screening. For this purpose, it does matter whether the collection of intersections return the answer effectively. The number of possible sets is huge, such as a group of candidates with special job skills or education levels, age, marital status, years of work, hobbies and so on. Third, the statistics also provide the factual basis for enterprises to establish a talent database, and the HR can establish a talent tracking mechanism through the results of the inquiry.

Applications similar to the previous one are abundant in practice. For instance, consider a tax database, where each category is a city, an item is the amount of tax paid by an individual in that city between January 1, 2016 and December 31, 2016. Then we can use the

multi-dimensional categorical range count query to "find out how many people will pay at least 8000 RMB in Beijing, Shanghai, Guangzhou and Shenzhen for the period January 1, 2016 to December 31, 2016, respectively."

In spite of its vast importance in reality and fundamental nature in database systems, surprisingly, the MCRA problem has not been studied before to our knowledge. Motivated by [2], we present the first work to address MCRA problem. It should be noted that the category range aggregation problems mentioned in this paper and the categorical data [3,4] are two different problems.

1.3 Contributions and Organization

Our main contributions can be summarized as follows:

- We describe an internal and external memory hybrid index structure that addresses MCRA problem. This hybrid index structure makes it much easier to build complex indexes than the disk pages and the cache memory data structures. Compared to the memory index structure, it can maximize memory savings, and store more data.
- We strictly certify the MCRI-tree's time and space complexity, and effectively answer any query even in the worst case. This feature is especially important in the environment where it is crucial to impose a hard bound on the maximum response.
- We did extensive experiments using synthetic and practical datasets to demonstrate that MCRI-Tree is much more efficient at query than existing solutions.

The rest of the paper is organized as follows. Section 2 describes the work that has been done to solve this problem in two special cases (One-dimensional and unclassified) and other related index techniques. In section 3, we proposed a data structure named MCRI-Tree, which can make more use of memory space to further improve I/O efficiency to solve MCRA problem. Section 4 validates the practical efficiency of our techniques with experiments. Finally, Section 5 concludes this paper.

2. Related Work

The earliest database index technology was B-Tree, which was an external memory data structure used to solve information index, such as file name in file system. After the B + tree proposed, all the data is stored in the leaf node, and the index structure is located at the non-leaf nodes. This technology used to accelerate range query [5]. Due to the superior randomness of B + trees and the performance of range queries, it has become the technology of choice for all commercial databases, even for distributed databases [6]. There are many literatures on B + tree optimization, such as concurrency control [7], targeted optimization of new hardware like Solid State Drive (SSD) [8,9,10,11] and so on.

Range aggregation query is an old problem. For example, this problem can be solved by aB-Tree [12] in the time of $O(log_B N)$. Y.Tao et al[2] proposed the bundled range aggregation problem, which is also the problem of categorical range aggregation in this paper. And they proposed an aBB-Tree to solve this problem. It's core idea is to create an aggregation result index page on a non-leaf node for each category. The index page records each sub-node of this non-leaf node in each category for the result of the gathering. The results of each category corresponding to the same child node are stored in contiguous positions. In order to support categorical range aggregation query, the aBB-Tree is modified based on aB-Tree as follows. First, change the index result to be stored in a new page. In order to reduce the number of I/O, it is optimized to be arranged in the order of child nodes and incrementally stores the

aggregation result of each child node so that it can be read twice to calculate the aggregation result of the middle child nodes, then reduce to $O(log_B N)$ I/Os. Second, in order to improve the uniform I/O of the update operation, one patch is added for each non-leaf node page so that the original $O(blog_B N)$ I/Os is reduced to $O(log_B N)$ I/Os.

In the case of multi-dimensional, Papadias et al. proposed aR-Tree [13] to solve the problem of multi-dimensional range aggregation (*that is, the special case of multi-dimensional range categorical aggregation query proposed in this paper with only one category*). The aR-Tree can be understood as an index structure for storing aggregated values in each node of the R-Tree. Since aggregation results only require O(1) space for storage without classification, the aR-Tree can answer the result of a multi-dimensional aggregation query within $O(log_B N)$ I/Os.

Although the aBB-Tree efficiently solves the CRA problem, there is still room for improvement. Initially, it is applied only to solving CRA problem, but can't for MCRA problem. In practical systems, the scope of the query is often more than one dimension, compared with the Multi-dimensional categorical range aggregation query more meaningful. Furthermore, because its aggregation result pages are arranged according to the order of the first leaf nodes and then sorted according to the order of the first leaf node. If the new category is dynamically added during the update, the aggregation result page of the whole tree needs to be updated, which is luxurious. In addition, the aR-Tree can not be directly extended from multi-dimensional range query to support multi-dimensional categorical range query.

As mentioned above, existing aggregation indexes currently have some problems, including the inability to efficiently solve MCRA problem, and inability to dynamically increase the categories, and the low utilization of space. In order to overcome these bottlenecks, the most intuitive and natural idea is to put the aggregation result page in memory. Driven by Moore's Law, the memory price is getting lower and lower, and now the PC usually has more than 4G of memory. Therefore, it is less costly for the aggregation results and non-leaf nodes stored in memory. When the scale of data is within a certain range, it is possible to store non-leaf nodes in memory. In this case, the memory read and CPU processing time should be reduced as much as possible to prevent bottlenecks. In order to reduce the memory read time, we should make full use of the CPU internal cache[14]. In general, the size of a single node for B-Tree and R-Tree is about 4KB, and the total size of the cache is only a few MB. Without proper optimization, the efficiency of these index structures will be reduced at the cache level.

Starting with B-Tree, people have already used cache to temporarily store pages in memory, which greatly reduces I/O. The Buffer-Tree[15,16] is the first index structure to have a separate cache on each node, which collecting small-scale written operation and the contents of cache will be written back to the node when the cache space of node is full. But doing this does not dynamic enough, it requires large cache space, and few other nodes are rarely written. Collecting the cache not only takes up memory but also has a negative impact on query performance. Therefore, some scholars have proposed LA-Tree[17], which can better adapt to different loads than Buffer-Tree. As mentioned earlier, this idea has also been adopted in the aBB-Tree to reduce the update overhead. In addition, the most representative work are CSS-Tree [14] and CSB+-Tree [18]. The idea of CSS-Tree is to maximize the cache hits as much as possible so that the nodes on the route traversing the B+-Tree are in the same or adjacent memory area. However, the CSS-Tree does not support the dynamic update of the data. In addition, lower space utilization is caused by stored pointers of child nodes. The CSB+-Tree reduces the number of pointers to one by placing the memory area of the child node of the same node in the same area. Due to the child nodes in a region, the cost of updating

operations greatly reduced. However, this design led to a decreace in search performance of CSB+-Tree relative to CSS-Tree. In short, the performance of CSS-Tree and CSB+-Tree relative to B+-Tree has greatly improved. However, their optimizations are for one-dimensional keys, and none of them can be directly extended to multidimensional.

Many scholars have also studied the new index structure, such as LSM-Tree [19], Fraction Tree [20,21] and FD-Tree [22]. Their main idea is to place part of the index structure in high speed but expensive storage media, such as non-volatile memory, and other new hardware, and the other part is placed in slow speed but inexpensive storage media, such as a hard disk drive (HDD). Also worth mentioning is the bLSM-Tree[23], which uses Bloom Filter technology to make a preliminary determination of the existence of data. Using this technique, hot and cold separation can be achieved by storing the data with higher access frequency (ie, “hot” data) and the data with lower access frequency (ie, “cold” data) separately in different media. For each query, first of all, check the hot data area, and then check the cold data area. We can easily determine whether the data exists in the hot area, which greatly speeds up the query. However, neither Fractional Cascading nor Bloom Filter is suitable for multi-dimensional situations and can not be used as multi-dimensional index.

In recent years, with the popularity of solid state drives (SSDs), many scholars have done a great deal of researches to optimize the performance of index structures on these new hardware. Such as Bw-Tree [24], which sets the conventional node in memory. In addition, it sets aside a region used to cache the operation of the node. This kind of caching mechanism has two roles in Bw-Tree. The first is to reduce the number of memory written because the area can be cached without swapped. The second is to make sibling nodes in the memory area as adjacent as possible, in order to improve the cache hit rate. However, the Bw-Tree also does not apply to multidimensional.

Multidimensional indexing has two main access models, one for point access and the other for spatial access [25]. In this paper, multi-dimensional refers to the point access, such as [26-28] , and other work are multidimensional query indexing problems, as well as in recent years[29-32]. However, these researches have nothing to do with this paper, we will not repeat.

3. Indexing Structure Design

Reviewing previous work, many scholars have made useful researches for the efficient index structure. However, there are few researches on multidimensional index structure. In this paper, we proposed a new index structure named Multidimensional Range Indexing-Tree (MRI-Tree), correspondingly its one-dimensional version named RI-Tree, and its version with categories indexing is named Multidimensional Categorical Range Aggregation Indexing -Tree (MCRI-Tree). The MRI-Tree can perform multidimensional query and update operations at O(1) I/Os. The MCRI-Tree adds an aggregation index to support MCRA query. In this section, we will first introduce MRI-Tree and then MCRI-Tree.

3.1 MRI-Tree

The MRI-Tree is a memory-disk hybrid index structure, whose memory layer to store the index structure, and the disk layer to store the leaf nodes. **Fig. 1** has shown a two-dimensional MRI-Tree. The white color is a dimension, the gray color is another dimension, and the page stores specific data points. Wherein each data point satisfies an interval corresponding to a node whose value corresponds to all the corresponding dimensions on the path from its parent

node to the root node. Formally, suppose the total dimension be d, N_i^j represents a node with a label of i and a dimension of j, and the interval that is saved in the node is set to $[s_{N_i}, e_{N_i}]$. Let N_1^1 be the root node and N_2^2, N_3^2 be the child nodes of N_1^1. P_i is the page that N_i stores in external memory when N_i is a leaf node. It is a sequence of up to B mappings from a

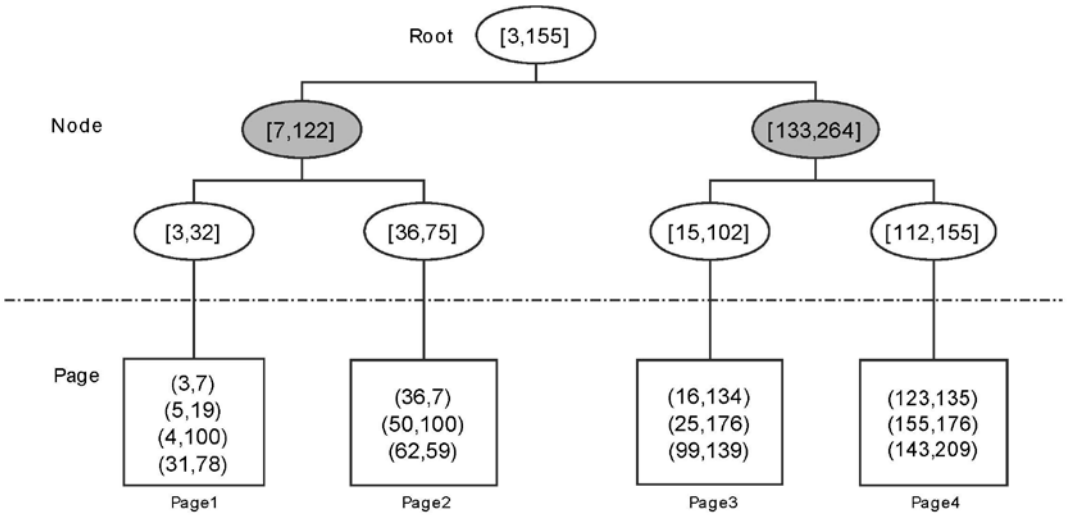


Fig. 1. An Example of the MRI-Tree

d-dimensional vector to a value of any type. We define $P_{ik}= (p_{ik}^{(1)},p_{ik}^{(2)},...,p_{ik}^{(d)})$ as the key of the k-th mapping pair stored in the sequence P_i, and V_{ik} is corresponding value. The path from N_1^1 to N_i is defined as the sequence $S_i = \{N_1^1,N_{i_1}^2,...,N_i\}$. The definition operator $Dim(N_i)$ is the dimension segmented by the node N_i, that is, $j = Dim(N_i)$.

In addition, the left child node of N_i is denoted $L(N_i)$. If the left child of N_i does not exist, $L(N_i) = \phi$. The corresponding the right child node of N_i is denoted $R(N_i)$, then the MRI-Tree can be defined as follows:

Definition 3 *Properties of the MRI-Tree: In the MRI-Tree, for each value $P_{ik}^{(j)}$ of each vector P_{ik} in each page P_i and the path sequence S_i corresponding to P_i, satisfy the following equation:*

$$\forall c_k \in S_i \;\; s.t. \;\; Dim(c_k) = j, p_{ik}^{(j)} \in [s_{c_k}, e_{c_k}] \tag{3}$$

Furthermore, for all i, the MRI-Tree also has the following properties:

- If $L(N_i) = \emptyset$, then $R(N_i) = \emptyset$.
- $\max\ L(N_i) \leq \min\ R(N_i)$. Specifically, the minimum value of the range of the right child node is not less than the maximum value of the range of the left child node. This interval may be referred to as the **middle interval** of the node.
- $d > 1$, $Dim(L(N_i)) \neq Dim(N_i)$, In any case, $Dim(R(N_i)) = Dim(L(N_i))$. In other words, the left and right child nodes should be in different dimensions with the father node, and the dimensions of the left and right child nodes are the same.
- $\forall\ i$, $S_i \neq \emptyset$, moreover, $N \in S_i$. That is, each page P_i must correspond to a page N_i in memory, and N_i must be located in a sub-Tree where N_1 is the root node.

Fig. 1 illustrates an example, the data point (36,7) in the page corresponds to the first dimension of the interval [3, 155] and [36, 75], the second dimension of the interval [7, 122]. Obviously, the value 36 of the first dimension and the value 7 of the second dimension respectively fall within the three intervals, and other properties are also satisfied.

Theorem 1 *The number of nodes in the MRI-Tree: Let N be the total number of elements. B is the maximum number of elements that can be stored in a page, then the number of nodes in an MRI-Tree memory is $O(N/B)$.*

Proof According to the properties of MRI-Tree, a page corresponds to a node, since the number of pages is $O(N/B)$, the number of leaf nodes is $O(N/B)$. It can be seen that the total number of nodes is $\sum_{i=0}^{logN/B} \frac{O(N/B)}{2^i} < 2\mathrm{O(N/B)}$ based on the binary-tree, that is $O(N/B)$. ■

According to the properties of binary-tree and the **Theorem 1**, the average height of an MRI-Tree is $\Theta(log\frac{N}{B})$. Assuming that the key of the data point to be operated is $q=(q_1, q_2, ..., q_d)$, the basic operation of the MRI-Tree is as follows:

Query. According to the **Definition 3**, start from the root node N_1^1. Suppose the current query node is N_i, if $q_{Dim(L(N_i))} \in L(N_i)$, let the current node become $L(N_i)$, otherwise, if $q_{Dim(L(N_i))} \in R(N_i)$, the node becomes $R(N_i)$, if not, the return does not exist. Then loop

Algorithm 1 MRIInsert

Input:

N_i : The current node, q : The key to wait for the inserted element, V : The value of the element to be inserted, $H(N_i)$: The height of node N_i.

Output:

The node after the element is inserted. If the insertion fails,(due to the existence of the key, the lack of space, etc.) then return ϕ.

if N_i is the leaf node **then**
  if $|P_i|+1 > B$ **then**
    MRISplit(N_i)
    $H(N_i) \leftarrow H(N_i)+1$
    return MRISplit(N_i, q, V)
  else
    $p_i \leftarrow p_i \bigcup (q \rightarrow V)$
    return N_i
  end if
end if
if $q_{Dim(L(N_i))} \le \max L(N_i) \vee (q_{Dim(L(N_i))} \in [e_{L(N_i)}, s_{R(N_i)}] \wedge H(L(N_i)) > H(R(N_i)))$ **then**
  $\mathrm{Re}\,sult \leftarrow MRIIsert(L(N_i), q, V)$
  $H(N_i) \leftarrow H(L(N_i))+1$
  retrun $\mathrm{Re}\,sult$
else
  $\mathrm{Re}\,sult \leftarrow MRIInsert(R(N_i), q, V)$
  $H(N_i) \leftarrow H(R(N_i))+1$
  retrun $\mathrm{Re}\,sult$
end if

executes the above steps until $L(N_i) = \phi$. Meanwhile, in the P_i to find whether the existence of q, if there is, then return the corresponding value of q, if not, return the corresponding flag. It can be seen that the operation is mainly carried out in memory, only need to read 1 page in external memory to find leaf node, so the time complexity of I/O is O(1).

Insertion. Algorithm 1 describes this recursive operation. Let V be the value corresponding to q. From the root node N_1^1, the relationship between $q_{Dim(L(N_i))}$ and the intervals $L(N_i)$, $R(N_i)$ should be judged for each operation, which is similar to the query operation. If $q_{Dim(L(N_i))} \in L(N_i)$, recursively insert the element into the left sub-tree. If $q_{Dim(L(N_i))} \in R(N_i)$, the elements should be recursively inserted into the right sub-tree. If neither is true, we need to consider whether this node is placed in the left sub-tree or right sub-tree. If $q_{Dim(L(N_i))} \in e_{L(N_i)}$ or $q_{Dim(L(N_i))} \ni s_{R(N_i)}$, that is not in the middle interval, then directly insert into the corresponding sub-tree. If the nodes are in the middle, we need the right approach to prevent sub-tree tilting. Here, we calculate and compare the heights of the left and right sub-tree, and put the node into the lower sub-tree, so as to achieve a balance-tree. If only one is true, it can be placed directly on the other side.

Example. To illustrate, we assume that point P(34,118) is inserted into the MRI-Tree, which can be seen in Figure 1. According to Algorithm 1, first, determine whether the first dimension of point P(34,118) is within the root node N(3,155). If so, continue, and then determine which child node P belongs to, and the second dimension (34,118) of P is located at (7,122). Next, determine where P should be inserted into the page, and we find that 34 does not belong to any one but is located in the middle interval by comparison with the range of values of the left and right leaf nodes. In order to insert P into the page, we need to compare the size of the corresponding page of each leaf node to obtain a smaller one. In order to balance the tree, we insert P into Page2, and finally update the value range of the corresponding node of the page. Replace (36,75) with (34,75).

Algorithm 2: MRISplit

Input: N_i : The current node

1: **begin**
2: If d_i has the same value, remove d_i in the total dimension.
3: $d_f \leftarrow f(N_i)$
4: Using the quick select algorithm, and select the median M of all $p_{ik}^{d_f}$ stored in P_i
5: $P_{left} \leftarrow ((p_{ik} \rightarrow V_{ik}) \mid p_{ik}^{d_f} < M)$
6: $P_{right} \leftarrow ((p_{ik} \rightarrow V_{ik}) \mid p_{ik}^{d_f} > M)$
7: $N_{left} \leftarrow [\min p_{left}^{d_f}, \max p_{left}^{d_f}]$
8: $N_{right} \leftarrow [\min p_{right}^{d_f}, \max p_{right}^{d_f}]$
9: $L(N_i) \leftarrow N_{left}$
10: $R(N_i) \leftarrow N_{right}$
11: **end**

If the current node is a leaf node, insert the element into the page P_i, that is, read the page and place the element at the end of the page, and then write the page. The time complexity of

reading and writing a page is O(B). If the size of the inserted page exceeds B. First, we should split operation, and then insert into the corresponding sub-node. The split operation has shown in **Algorithm 2**. It is similar to a quick selection algorithm, to find the median of a dimension, and to place elements less than it to the left, larger than it to the right.

In **Algorithm 2**, a dimension selection function $f(N_i)$ is introduced, which inputs the current node N_i and returns a dimension $d_f \in [1, d]$ and $d_f \neq Dim(N_i)$. This function for different scenarios can have different ways. However, we need to avoid the case that all the data points have the same value for the selected dimension. If the dimension has the same value, the split operation will not be executed. Therefore, it is removed from the candidate dimension at the beginning of the algorithm. In general, the $f(N_i)$ can be taken as $f(N_i) = ((Dim(N_i) + 1) mod\ d) + 1$. Concretely speaking, take the current node's dimension number plus one and then take the remainder of d, and the remainder plus one to ensure that it falls into the interval [1, d]. Thus it can be seen that this process only needs to be read once and written twice, and the most consuming is the quick selection operation, whose time complexity is O(B). Therefore, the time complexity of the entire split operation is O(B), the I/O complexity is O(1).

Finally, no matter which sub-tree is placed, after the recursive insertion of this layer is completed, the current node's range needs to be updated to maintain that it still satisfies the properties of the MRI-Tree. Owing to at most one split operation, one read operation and one write operation, the I/O complexity of the entire insertion operation is O(1) and the time complexity is $\mathrm{O}(\mathrm{B}+\log\frac{N}{B})$.

Deletion. The delete operation of MRI-Tree is similar to the insertion operation, specifically, recursively find the page and the node where the element is located, then removing the element from the page. After deletion, if the number of elements in a sub-tree is less than a certain threshold, all the pages in the sub-tree are merged. The complexity of this operation is the same as the insertion operation.

3.2 The MCRI-Tree

The MRI-Tree introduced in the previous section is a special case of the MCRI-Tree that does not contain an aggregation index. However, the MCRI-Tree needs to provide an aggregation function G. In this section, the aggregation function is defined as COUNT in the SQL.

Formally, the MCRI-Tree is a -Tree with an aggregation result mapping sequence $A(N_i)$ on each node N_i. In addition, there is a bitmap $B(N_i) = \{b_{i1}, b_{i2}, ..., b_{id}\}$ to hold the various categories in the grouping column. Assuming that the number of categories is *C*, and each element in the sequence $A(N_i)$ can be denoted by $c_{ij} \rightarrow a_{ij}$. i.e., a_{ij} represents the aggregation result of the c_{ij} categories at the *i-th* node, meanwhile, $|A(N_i)| \leq C$, $c_{ij} \in [1, C]$.

Example. To illustrate, a three-dimensional MRCI-tree is established based on **Table 1**, as shown in **Fig. 2**. The nodes of each layer correspond to different dimensions. (i.e., entry-time, age and salary from top to bottom). Here, we assume that the dimension of grouping statistics is the *city* column. In other words, we need to group by the city column. From table 1, we can see that there are three categories of *city* columns, i.e., *BJ, MH, NYC*. To keep the names of the categories in order, we add the corresponding bitmap $B(N_i)$ at each node. For instance, *BJ*, *MH* and *NYC* at node N_4 are arranged and saved in the bitmap. In addition, there is an aggregation mapping sequence $A(N_i)$ on each node N_i for counting statistics of various categories, and the counting results correspond to the names of each category. e.g., for N_2, *BJ*, *MH* and *NYC* are counted as 2, 2, 1 respectively.

The basic operations of MCRI-Tree are similar to MRI-Tree. However, it involves the issue of updating the aggregation index. In addition, the most important is how to quickly respond to the range aggregation query and the category range aggregation query. The following two subsections will introduce the basic operations and aggregation queries, respectively. The last two subsections will analyze MCRI-Tree's I/O efficiency and internal and external memory space efficiency.

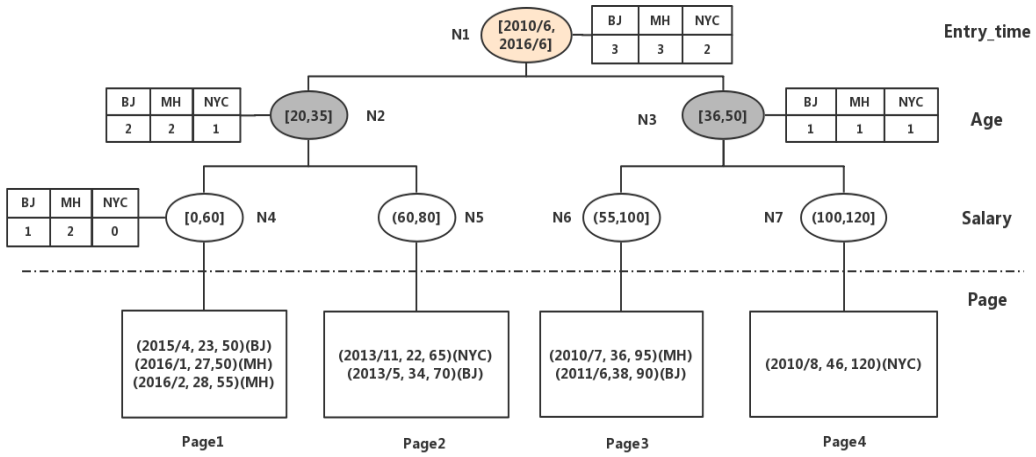


Fig. 2. An Example of the MRCI-Tree

3.2.1 The Basic Operations

Compared to the MRI-Tree, only the update operation has changed, so we only describe how to update the aggregation index.

Insertion. In MRI-Tree, the height of the node is updated after the insertion operation returns. At the same time, the aggregation result is updated, and add 1 to the count of the category corresponding to the newly inserted element. If the category of the newly inserted elements does not exist, the aggregation result is inserted into the category according to the category sequence number, and the count can be set to 1. In addition, through the dimensional bitmap, do *OR* operation to the child node's bitmap, in any case, the time complexity of updating for the aggregation result sequence is O(C).

Deletion. Contrast to the insertion operation, the category counts of the deleted elements subtract 1. In the case of 0, it can be removed from the aggregation result sequence. In the case of merge, we only need to free up the memory space of the aggregation sequence of deleted nodes.

3.2.2 The Aggregation Query

Algorithm 3 describes range aggregation query. q is the interval to be queried, and satisfies $q_i = [q_{i_{min}}, q_{i_{max}}] \in q$. In short, each dimension is an interval, q can also be called the query hypercube. First, the relationship between $q_{Dim(N_i)}$ and N_i should be determined. If there is no intersection, return directly to the empty set. If $N_i \subseteq q_{Dim(N_i)}$, return the corresponding to aggregation result of this node.

Example. As can be seen from **Fig. 2**, the aggregation results of corresponding dimensions are saved at each node in MCRI-Tree. Suppose our query dimensions are restricted to Entry_time and Age. The grouping dimension selects the city column and executes the following SQL.

SELECT COUNT(*) **FROM** Table 1.
WHERE Entry_time > '2010/6' **AND** Age =< '35'
GROUP BY City

We can get the answer in the aggregation result of node N_2.

If N_i is a leaf node, gather together all the key p_{ik} corresponding to the values V_{ik} in P_i, falling within q range, using the aggregation function σ and then return; otherwise, recursively calculate the aggregation result in the left child and right child, and merge the two aggregation results, and generate a new sequence.

Algorithm 3 MCRIAggregate

Input:
N_i: The current node, q: Interval to be inquired
Output:
$\sigma(N_i, q)$: Aggergation result of q in sub-Tree of node N_i

```
1: Begin
2:    if q_Dim(N_i) ∩ N_i = φ then
3:       return φ
4:    end if
5:    if N_i ⊆ q_Dim(N_i) then
6:       return A(N_i)
7:    end if
8:    if N_i is the leaf node then
9:       return σ(V_ik | p_ik ∈ q)
10:   end if
11:   return MCRIAgg(L(N_i), q) ∪ MCRIAgg(R(N_i), q)
12: End
```

3.2.3 I/O Complexity Analysis

First of all, we analyzed the complexity of the MCRI-Tree update operation. From the perspective of I/O efficiency, the complexity of the two update operations is still O(1) relative to the MRI-Tree. From the computational time perspective, the time complexity of the two operations is changed from $O(Blog(N/B)$ of MRI-Tree to $O(max\{C, B\}log(N/B))$.

Theorem 2 *The I/O complexity of MCRI-Tree is:*

$$O(d(\frac{N}{B})^{\frac{d-1}{d}}) \tag{4}$$

Proof Intuitively, the I/O count is the number of the smallest hypercube of the hypercube to be queried "cut" around each point in the page. The condition for the two hypercubes to form a cut relationship is that the intersection of the intervals on each dimension is not null, and there is at least one dimension where the intersection of the cubes to be inquired and the corresponding cube interval of the page is not equal to the interval corresponding to the page. Formally, we make the assumption that the hypercube to be queried is $r = [s_1, e_1], [s_2, e_2], ..., [s_d, e_d]$, and the set of cubes intersecting with the hypercube to be queried are shown as follows:

$$I = \{P_i \mid \forall j \in [1,d], [s_{ij}, e_{ij}] \bigcap [s_j, e_j] \neq \phi \wedge \exists j \in [1,d], [s_{ij}, e_{ij}] \bigcap [s_j, e_j] \neq [s_{ij}, e_{ij}]\} \tag{5}$$

This set is equivalent to the following set

$$I_1 \bigcup I_2 \bigcup \ldots \bigcup I_d \tag{6}$$

Where $$I_k = P_i \mid \forall j \in [1,d], [s_{ij}, e_{ij}] \bigcap [s_j, e_j] \neq \phi \wedge [s_{ik}, e_{ik}] \bigcap [s_k, e_k] \neq [s_{ik}, e_{ik}] \tag{7}$$

It can be seen that every hypercube in each set of I_k must intersect the surface of the hypercube r in k-th dimension, otherwise it can not satisfy the condition:

$$[s_{ik}, e_{ik}] \bigcap [s_k, e_k] \neq [s_{ik}, e_{ik}] \tag{8}$$

Therefore, the upper bound of the total number of hypercubes intersecting is $kmax_{j\in[1,d]}\{|I_j|\}$. In order to compute $max_{j\in[1,d]}\{|I_j|\}$, intuitively, it can be calculated from the maximum number of cubes in each dimension. The page number of external memory is $\frac{N}{B}$, so the internal nodes split $\frac{N}{B}$-1 times. Suppose the height of -Tree is uniform, the number of splits of each dimension is about $2^{\frac{log\frac{N}{B}}{d}}$, which is obtained the numbers of lines by hyperplane cutting the dimension when all other dimensions are fixed. The surface of a dimension of a hypercube is a (*d*-1)-dimensional hypercube. Therefore, the available number of hypercubes is $2^{\frac{log\frac{N}{B}}{d}(d-1)} = (\frac{N}{B})^{\frac{d-1}{d}}$. Each of these surfaces has these hypercubes, so the upper bound of the total number of hypercubes is $O(d(\frac{N}{B})^{\frac{d-1}{d}})$. ■

We use one-dimensional and two-dimensional cases to show the correctness of the results:

- d = 1, the MCRI-Tree aggregation query only needs to read two pages from the external memory. Intuitively, each page in which the interval does not intersect, while the query range is only a starting and an ending point. In the worst case, the starting point falls within the range of a page, and the ending point falls within the interval of another page. Therefore, only the two pages can be read, and the results of the other pages are stored in the memory node. For $d = 1$, the I/O complexity of MCRI-Tree is O(1).
- d = 2, each page falls within a two-dimensional interval, and these intervals do not intersect with each other. The query interval is a rectangle on R^2 with four edges, each of which is a rectangular surface, which can be cut to obtain $(\frac{N}{B})^{\frac{1}{2}}$ rectangles. Intuitively, that is, $4\times$ cut the most rectangular edge to meet eq. (5).

3.2.4 Space Complexity Analysis

Owning to the MCRI-tree is a memory-disk hybrid index structure, the memory stores an index structure, and the external memory mainly stores a specific data items N. Therefore, the size of the external memory occupied by MCRI-tree is O(N), and the index part is equal to the number of nodes in the memory. According to the properties of MRCI-tree, one page corresponds to one node. The number of pages is O(N/B), where B is a page size. Therefore, the memory occupied by the index part is O(N/B).

3.3 Summary

This section mainly presents two new data structures which are MRI-Tree and MCRI-Tree. Among them, the MRI-Tree is used for pointing query and various update operations in O(1) I/Os. In addition to the features of MRI-Tree, the MCRI-Tree also supports $O(d(\frac{N}{B})^{\frac{d-1}{d}})$ I/Os, and solves the categorical range aggregation query problem. The experimental results will be shown in the next section to prove this.

4. Experimental Design and Result

4.1 Experimental Design

This section will include a number of comparative and confirmatory experiments, which are listed below:

- Compare the efficiency of MCRI-Tree aggregation query to prove its better performance. Since the problem of multi-dimensional categorical range aggregation query is a new problem in this paper, the categorical range aggregation has only one previous work, the contrast experiment is limited to a single-dimensional case. In multi-dimensional cases, the space required for aR-Tree index is too large and the query speed is too slow. The performance of MCRI-Tree and aR-Tree can not be directly compared in this experiment, so we choose MRI-Tree (also known as raw MCRI-Tree) and MCRI-Tree for comparison.
- The memory occupancy of MCRI-Tree is recorded according to the change in the size of the dataset and the number of dimensions to verify the correctness of the space complexity.
- The external memory occupancy of MCRI-Tree is recorded according to the change in the size of the dataset to verify the correctness of the space complexity.

The above experiment was carried out on a server equipped with Intel Xeon E5620 @ 2.4GHz CPU, which memory size is 32GB, the external memory space uses a Western Digital 7200 RPM hard drive, the operating system is Windows Server 2012. All experimental code is based on C ++ 11 standard, compiled with the Microsoft Visual C ++ 12.0 compiler, and turns on the -O2 optimization switch.

4.1.1 Dataset

The dataset used in the experiment is divided into two types. First of all, we generated datasets where items' key and weight are uniformly distributed in $[0, 2^{30}\text{-}1]$ between the use of uniform distribution generated random positive integer.

Secondly, we also made an experiment with a real dataset called $S\&P500$, which contains the daily trading volumes of every stock in the $S\&P500$ index from 16 Jan. 2002 to 13 Jan. 2017. Each stock is a category (*i.e., totally b = 500 categories*), in which an item is of the form (*date, open, high, low, close, volume*), where date is the *items* key, and others is its weight. The total number *N* of items (*of all stocks*) is 1.89 million. The query mainly depends on a given parameter $\rho \in (0, 1]$, ρ is used to represent each dimension of the query interval accounted for the proportion of the size of the entire interval.

4.1.2 Parameters Set

Table 2. Parameter setting

Parameter	Symbol	Range	Default
Category number	b	(10, 1000)	50
Dimension number	d	(2, 10)	2
Index size	N	(16, 1024)	512
Page size	B	(4, 512)	4KB
Generate query parameters	ρ	(0.1, 0.9)	0.25

In this experiment, each index structure consists of four parameters, which are category number b, dimension number d, index size N and page size B. In addition, the query generation is controlled by the parameter ρ. In particular, when b = 1, it degenerates into a general range aggregation query. When d = 1, it degrades to a one-dimensional range aggregation query. When ρ = 1, it degrades to find the global aggregation result. The above parameter settings are shown in **Table 2**.

4.1.3 Experimental Content

In summary, this section includes the following experiments:

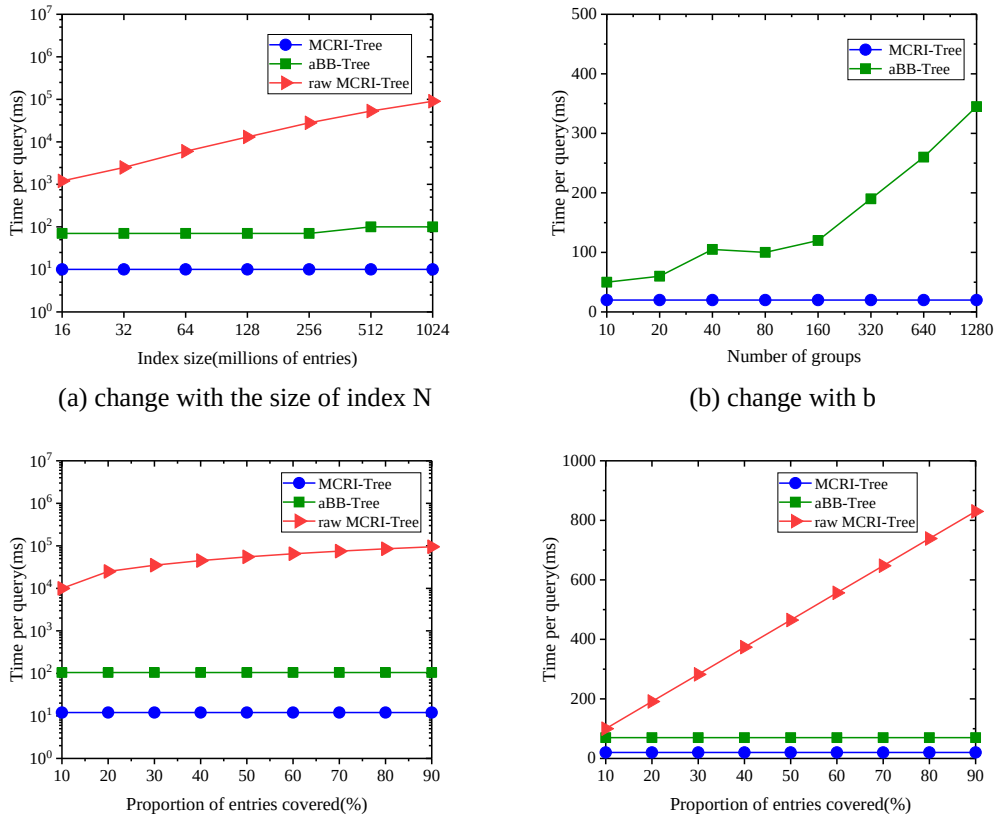
- The performance of the CRA query (d=1) is tested by synthetic dataset and the $S\&P500$ dataset. The evaluation indicator is the average elapsed time of each query, and the comparison between aBB-Tree and MCRI-Tree is done by adjusting the index size N, the number of category b, parameter ρ, to observe the performance of the two changes. It is worth mentioning that, in order to eliminate the disadvantage of aBB-Tree as a data structure with only external memory index, the total memory footprint of MCRI-Tree was recorded in the comparison experiment, and allocated aBB-Tree equivalent cache to cache pages in the external memory.
- Using synthetic dataset, we tested MCRA query (d>1) performance and compared MCRI-Tree with MRI-Tree. The evaluation indicator is the I/O counts of each query, where the average time is not used because the page size is the same and each access unit is a page. We adjust d, N, ρ, B, respectively, to observe the performance change of the both.
- Using synthetic dataset, we adjust N to test MCRI-Tree occupancy of internal and external memory, and then fix N, adjusting d, to observe MCRI-Tree performance change of memory.

4.2 Experimental Results

4.2.1 The CRA Query Experimental Results

Fig. 3 shows a comparison result of aBB-Tree and MCRI-Tree in CRA query. It can be seen that the result of MCRI-Tree is ahead of aBB-Tree in each test. **Fig. 3(a)** shows the change of the query average time with the size of index. It can be seen that the performance change of aBB-Tree and MCRI-Tree is not obvious as the index size grows. It is because of the scale of the experimental data, the height of aBB-Tree $O(log_B N)$ is almost unchanged. However, the I/O efficiency of MCRI-Tree is constant O(1), so the performance change of the both are not large. Nevertheless, the I/O complexity of MCRI-Tree is lower, so the performance of MCRI-Tree nearly doubled than aBB-Tree. **Fig. 3(b)** shows the change of query time with the number of categories b. It can be seen that MCRI-Tree always maintains a low level, not changing with the number of categories. However, aBB-Tree performance gradually deteriorates after the number of categories increases. Although the time complexity of aBB-Tree query aggregation result page is O(1), it is actually read an entry of length b sequentially from the external memory, instead of a real random I/O. The larger the b, the stronger the amplification effects. Because it is read sequentially, the bottleneck is the seeking time. Therefore, b is not a strictly multiple relationship when it increases to a larger size. **Fig. 3 (c)** and **Fig. 3 (d)** show a performance change of ρ. It can be seen that the performance of aBB-Tree and MCRI-Tree is obviously improved relative to MRI-Tree based on simple sequential scan. Meanwhile, MCRI -Tree continues to maintain a lead performance and the

performance of the both does not change with the interval size, which proves theoretically the correctness of the I/O efficiency.

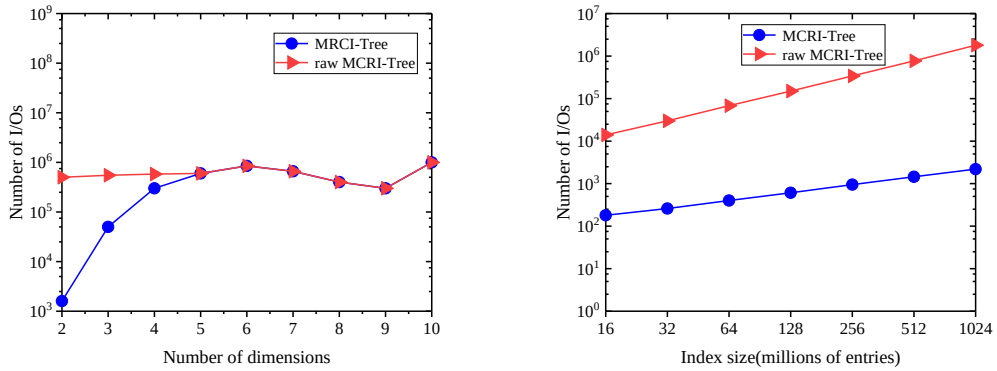


(a) change with the size of index N (b) change with b

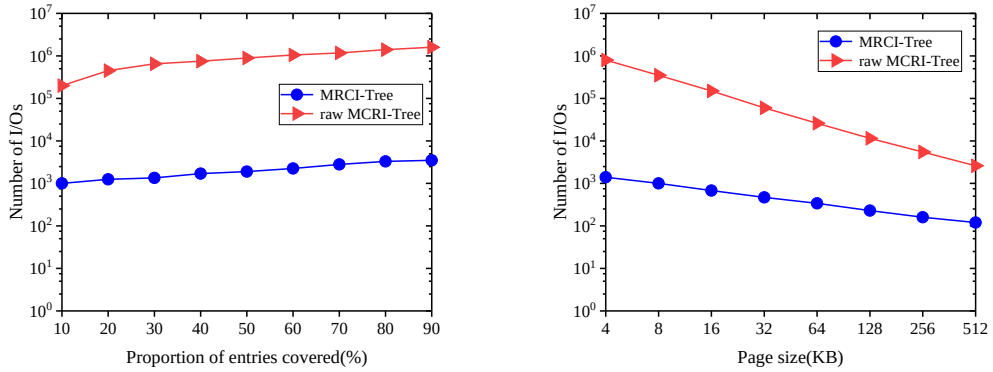
(c) change with parameters on synthetic datasets (d) change with parameters on S&P500 datasets

Fig. 3. Categorical Range Aggregation Query (CRA) performance

4.2.2 The MCRA Query Experimental Results



(a) Change with the number of dimension d (b) Change with the scale of dataset N



(c) Change with parameters ρ (d) change with the size of page B

Fig. 4. Multi-dimensional Categorical Range Aggregation Query (MCRA) performance

Fig. 4 shows the result of a MCRI-Tree test on a MCRA query, comparing it to MRI-Tree, in order to make a reference based on a simple sequential scan method. Here we compared the number of I/O directly. Since the index structure of MCRI-Tree and MRI-Tree is the same, we just compare I/O time. MCRI-Tree has a lot of superiority when the number of dimensions is less than 5, while the performance of MCRI-Tree and MRI-Tree is close at high dimensions. According to the I/O efficiency of MCRI-Tree demonstrated in the previous section, MCRI-Tree basically degenerates into a sequential scan when *d* is large, because almost every page needs to be read once. The experimental results of adjusting the size of the dataset are shown in **Fig. 4(b)**. It can be seen that with the change of the dataset scale, the performance change of MCRI-Tree is not obvious. This is because the change in scale is just a slight increase in $log\frac{N}{B}$, which is very small when x = 2.

Fig. 4(c) shows the efficiency change of ρ. MCRI-Tree change is still not obvious, however, the change of MRI-Tree is obvious. This is because the performance of MCRI-Tree changes little with ρ when the number of dimensions is small, while the performance of MRI-Tree degrades because it is a linear scan. Finally, an experiment to adjust the page size is shown in **Fig. 4(d)**, which the number of I/O decreases as the number of pages decreases, but the time of a single I/O will increase.

4.2.3 The MCRA Space Efficiency

Fig. 5 shows the space efficiency experiment of MCRI-Tree, and the memory space occupancy of MCRI-Tree is basically a linear function of the data scale. In the case of one billion data entries, the memory space occupancy is about 1.5GB. Considering index entries are on the order of magnitude, computers are often equipped with high memory capacity. In addition, it is also seen that the amount of space occupied by the counter portion is several times than the index portion, which also conforms to the spatial complexity analyzed above.

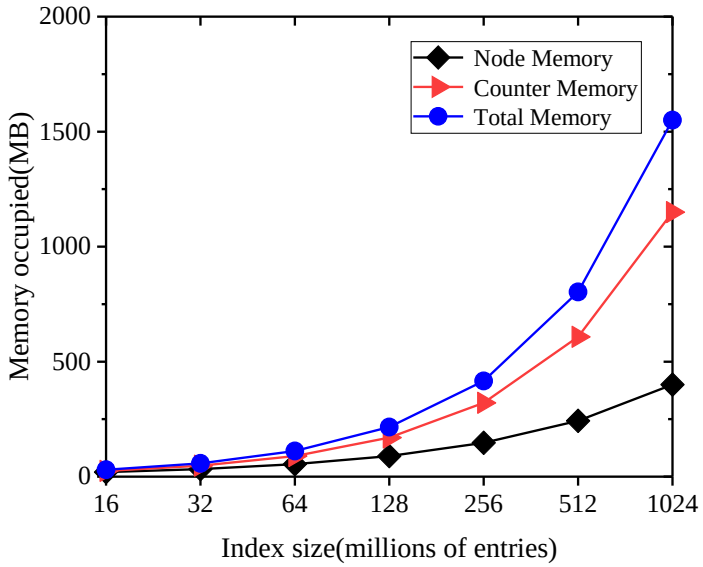


Fig. 5. Memory footprint of MCRI-Tree changed with dataset size

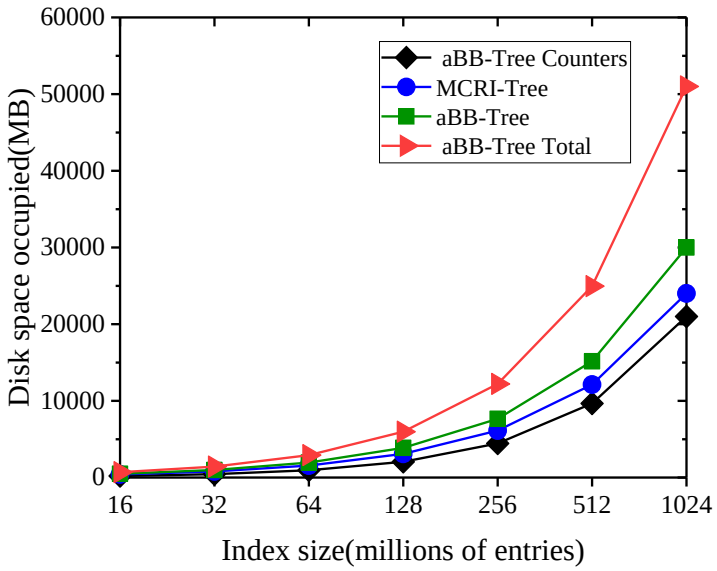


Fig. 6. External memory footprint of MCRI-Tree changed with dataset size

The experiments of the number of fixed entries, the page occupancy space and the number of the change in the dimension are shown in **Fig. 6**. Due to no change in the page size, resulting in a reduction in the number of entries stored per page, the decrease of B leads to the increase of $O(\frac{N}{B})$, whose relationship is correctly reflected in the figure. Finally, the statistics and comparison of external memory space are shown in **Fig. 7**. It can be seen that MCRI-Tree further improves the space utilization of the whole system compared with aBB-Tree.

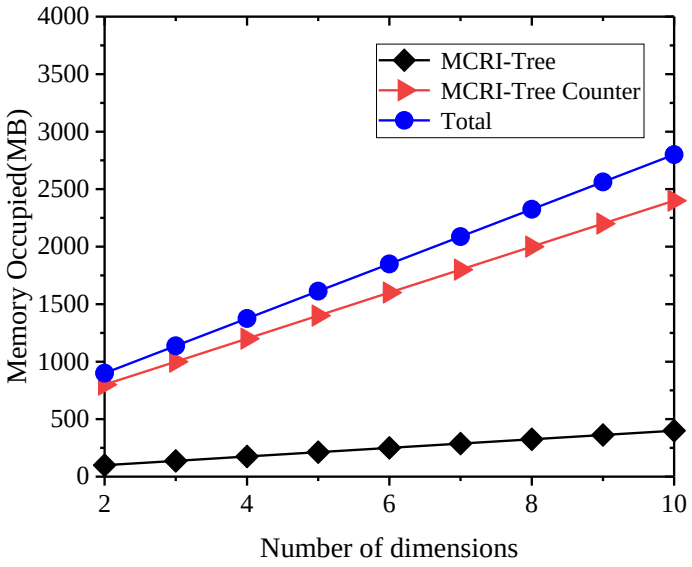


Fig. 7. Memory footprint of MCRI-Tree changed with the dimension

5. Conclusion

On the basis of the original aggregation query problem, the past work has proposed categorical aggregation query problem, and the corresponding aggregation index structure aBB-Tree is proposed to solve the query problem. However, categorical aggregation index has some limitations, which can't execute range aggregation query for multi-dimensional constraints. In order to solve this problem, we proposed a new index structure called MCRI-Tree. MCRI-Tree is based on the internal and external memory hybrid mechanism, which can answer the categorical aggregation query problem in $O(d(\frac{N}{B})^{\frac{d-1}{d}})$ I/Os, respectively while occupying $O(\frac{min\{d,B\}N}{B})$ of memory space and O(N) of the external memory space, where d is the number of dimensions, N is the total number of elements, B is the number of elements which can be stored in a single disk page. In particular, d = 1, only O(1) I/Os is needed, and $O(d\sqrt{\frac{N}{B}})$ I/Os is only required for d = 2. The result of the experiment proved this conclusion, and at the same time proved that space occupied by its internal and external memory was acceptable for a system, and illustrated its feasibility.

References

[1] Nekrich, Yakov, "Efficient range searching for categorical and plain data," *Acm Transactions on Database Systems*, vol. 39, no. 1, pp. 1-21, January, 2014. Article (CrossRef Link).

[2] Tao, Yufei, and C. Sheng, "I/O-Efficient Bundled Range Aggregation," *IEEE Transactions on Knowledge & Data Engineering*, vol. 26, no. 6, pp. 1521-1531, June, 2014. Article (CrossRef Link).

[3] S. Singh, C. Mayfield, S. Prabhakar, R. Shah and S. Hambrusch, "Indexing Uncertain Categorical Data," in *Proc. of IEEE Conf. on Data Engineering*, pp. 616-625, April 15-20, 2007. Article (CrossRef Link).
[4] N. Sarkas, G. Das, N. Koudas and A.K.H. Tung, "Categorical skylines for streaming data," in *Proc. of the 27th ACM SIGMOD International Conference on Management of Data, SIGMOD'08*, Vancouver, Bc, Canada, pp. 239-250, June 9-12, 2008. Article (CrossRef Link).
[5] D. Comer, "Ubiquitous B-Tree," *Acm Computing Surveys* vol. 11, no. 2 pp. 121-137, 1979. Article (CrossRef Link).
[6] M.K. AguileraW. Golab and M.A. Shah, "A practical scalable distributed B-tree," in *Proceedings of the Vldb Endowment*, vol. 1, no. 1, pp. 598-609, August, 2008. Article (CrossRef Link).
[7] C. S. Ellis, "Concurrent Search and Insertion in AVL Trees," in *IEEE Transactions on Computers*, vol. C-29, no. 9, pp. 811-817, Sept, 1980. Article (CrossRef Link).
[8] C.H. WuT.W. Kuo and L.P. Chang, "An efficient B-tree layer implementation for flash-memory storage systems," *Acm Transactions on Embedded Computing Systems*, vol. 6, no. 3, Article 19, July 2007. Article (CrossRef Link).
[9] H. Roh, S. Kim, D. Lee and S. Park, "As B-Tree: A Study of an Efficient B+-tree for SSDs," *Journal of Information Science & Engineering*, vol. 30, no. 1, pp. 85-106, January, 2014. Article (CrossRef Link).
[10] H. Roh, S. Park, S. Kim, M. Shin and S.W. Lee, "B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives," in *Proceedings of the VLDB Endowment*, vol. 5, no. 4, pp. 286-297, 2012. Article (CrossRef Link).
[11] P. JinP. Yang and L. Yue, "Optimizing B+-tree for hybrid storage systems," *Distributed & Parallel Databases* vol. 33, no. 3, pp. 449-475, September, 2015. Article (CrossRef Link).
[12] J. Yang and J. Widom, "Incremental Computation and Maintenance of Temporal Aggregates," *Vldb Journal*, vol. 12, no. 3, pp. 262-283, October, 2003. Article (CrossRef Link).
[13] D. Papadias, P. Kalnis, J. Zhang and Y. Tao, "Efficient OLAP Operations in Spatial Data Warehouses," in *Proc. of 7th Int. Symposium on Advances in Spatial and Temporal Databases* , Redondo Beach, CA, USA,Berlin, pp. 443-459, July 12-15, 2001. Article (CrossRef Link).
[14] J. Rao and K.A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," in *Proc. of 25th International Conference on Very Large Data Bases*, VLDB'99, pp. 78-89, September 07 - 10, 1999. Article (CrossRef Link).
[15] L. Arge, "The buffer tree: A new technique for optimal I/O-algorithms," in *Proc. of 4th Int. Workshop on Algorithms and Data Structures(WADS'95).* vol. 3, no.28, pp. 334-345, 1995. Article (CrossRef Link).
[16] L. Arge, "The Buffer Tree: A Technique for Designing Batched External Data Structures," *Algorithmica*, vol. 37, no. 1,pp. 1-24, 2003. Article (CrossRef Link).
[17] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao and S. Singh, "Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices," *in Proc. of the Vldb Endowment*, vol. 2, no. 1, pp. 361-372, January 2009. Article (CrossRef Link).
[18] J. Rao and K.A. Ross, "Making B+- trees cache conscious in main memory," *Acm Sigmod Record*, vol. 29, no. 2, pp. 475-486, June 2000. Article (CrossRef Link).
[19] P. O Neil, E. Cheng, D. Gawlick and E. O Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351-385, June 1996. Article (CrossRef Link).
[20] E.D. Demaine and M. Farach-Colton, "Cache-Oblivious B-Trees," *Siam Journal on Computing*. vol. 35, no. 2, pp. 341-358, 2005. Article (CrossRef Link).
[21] B.C. Kuszmaul, "A comparison of fractal trees to log-structured merge (LSM) trees," *Tokutek White Paper*, 2014. Article (CrossRef Link).
[22] Y. Li, B. He, R.J. Yang, Q. Luo and K. Yi, "Tree indexing on solid state drives," *Proceedings of the VLDB Endowment* , vol. 3, no. 1-2, pp. 1195-1206, September 2010. Article (CrossRef Link).
[23] R. Sears and R. Ramakrishnan, "bLSM: a general purpose log structured merge tree," in *Proc. of the 31th ACM SIGMOD International Conference on Management of Data*. SIGMOD'12. Scottsdale, Arizona, USA, pp. 217-228, May 20 - 24, 2012 Article (CrossRef Link).

[24] J.J. LevandoskiD.B. Lomet and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," in *Proc. of IEEE Conf. on Data Engineering*. pp. 302-313, April 8-12, 2013. Article (CrossRef Link).
[25] V. Gaede, "Multidimensional access methods," *ACM Computing Surveys (CSUR)*. vol. 30, no. 2 pp. 170-231, 1998. Article (CrossRef Link).
[26] M. Freeston, "A general solution of the n-dimensional B-tree problem," *ACM SIGMOD Record*. vol. 24, no. 2, pp. 80-91, 1995. Article (CrossRef Link).
[27] S. Nishimura, H. Yokota, "QUILTS: Multidimensional Data Partitioning Framework Based on Query-Aware and Skew-Tolerant Space-Filling Curves," in *Proc. of the 36th ACM International Conference on Management of Data*. SIGMOD'17. Chicago, Illinois, USA, pp. 1525-1537, May 14-19, 2017. Article (CrossRef Link).
[28] D.B. Lomet, "The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *Acm Transactions On Database Systems(TODS)*, vol. 15, no. 4, pp. 625-658, 1990. Article (CrossRef Link).
[29] B. Wang, Y. Hou, M. Li, H. Wang and H. Li, "Maple: scalable multi-dimensional range search over encrypted cloud data with tree-based index," in *Proc. of the 9th ACM symposium on Information, computer and communications security*, pp. 111-122, June 04-06, 2014. Article (CrossRef Link).
[30] G. Li, P. Zhao, L. Yuan and S. Gao, "Efficient Implementation of a Multi-Dimensional Index Structure Over Flash Memory Storage Systems," *Journal of Supercomputing*, vol. 64, no. 3, pp. 1055-1074, June 2013. Article (CrossRef Link).
[31] T. ZäschkeC. Zimmerli and M.C. Norrie, "The PH-Tree - a Space-Efficient Storage Structure and Multi-Dimensional Index," in *Proc. of the ACM SIGMOD international conference on Management of data*. SIGMOD'14, Snowbird, Utah, USA, pp. 397-408, June 22-27, 2014. Article (CrossRef Link).
[32] A. Vlachou, "Efficient RDF Query Processing using Multidimensional Indexing," in *Proc. of the 21st Pan-Hellenic Conference on Informatics*, Larissa, Greece, Article No. 44, September 28-30 2017. Article (CrossRef Link).

Jian Yang received the M.S. degree from North Minzu University, China, in 2015. He is currently as a PHD student in School of Computer and Communication Engineering, University of Science and Technology Beijing (USTB). His research interest includes massive data management, information retrieval and data quality management.

Chongchong Zhao is a professor and supervisor of PHD student. He received the Ph.D. degree from Northwestern Polytechnical University, Xi'an, China, in 2003. He is currently working at School of Computer and Communication Engineering, University of Science and Technology Beijing (USTB). His research primarily focuses on software engineering, high performance computing, data and knowledge engineering.

Chao Li is an associate professor. She received the Ph.D. degree from Tsinghua University, Beijing, China, in 2006. She is currently working at Information Technology Research Institute（RIIT）, Tsinghua University, Beijing, China. Her research interests are in the areas of data and knowledge engineering, cloud computing and data management.

Chunxiao Xing is a professor and supervisor of PHD student. He received the Ph.D. degree from Northwestern Polytechnical University, Xi'an, China, in 1999. He is currently working at Information Technology Research Institute（RIIT）, Tsinghua University, Beijing, China. His research primarily focuses on data and knowledge engineering, software engineering, cloud computing, the internet of things, and digital library technology.