International Journal of u- and e- Service, Science and Technology
Vol.8, No.4 (2015), pp.159-168
http://dx.doi.org/10.14257/ijunesst.2015.8.4.16

# An Efficient Index Structure for Range Count Query Based on Improved B-tree

Hu Jinglin and Qi Deyu

*Research Institute of Computer Systems at South China University of Technology*
*South China University of Technology*
*Guangzhou, P.R. CHINA*
*atlankie@hotmail.com*

***Abstract***

*With the development of many new technologies, the data grows much faster than before, and most of the data are stored in database. When the data amount in database is huge, it is needed to create appropriate index to keep the performance of database applications. This paper proposes a new index structure, namely MCRC-tree, to solve complicated range count query problem efficiently. Multiple experiments show that the performance of MCRC-tree is similar to that of traditional B-tree, to be specific, MCRC-tree can solve any range count query in $O(log_c N)$ I/Os, so is the complexity of insert and delete operations, here C means the disk page size, N is the total data amount.*

***Keywords:*** *index; B-tree; range count*

## 1. Introduction

Nowadays a lot of new way of information release such as blogs, social networks, LBS (location-based services), and new technology such as cloud computing and the Internet of things emerge constantly, which result that the data amount grows with unprecedented speed. The world has entered the era of big data. According to IDC the total data amount which was created and copied in 2011 is 1.8 ZB. The number of users enrolled in facebook is beyond a billion, and the members of taobao are about a half billion. These members create a large number of data every day through all kinds of network operation, such as logging, shopping, uploading new album etc. a large part of these data are stored in the database. When the data amount becomes enormous, it is a challenge to keep the performance of database applications.

Normally database applications all have essential statistical functions, one of which is range count query. In order to improve statistical efficiency one of useful methods is to create various efficient index [1-5]. A typical range count query is as following

*select count(*) from table_name where field1 between A and B*

The query can be easily solved through full table scan when the data amount is not big. Once the data amount become huge, full table scan is too time-consuming because of I/O operations. The key of query performance is to minimize I/Os. A lot of study have been made on the subject[6,7,8,9,10], these study have their own specific conditions, and can't be used to solve complex range count problems efficiently, next section we'll introduce this kind of range count query.

## 2. Problem Description

In many cases we can divide data into multiple classes naturally. For example considering a national traffic accident database, which records (date, city, accident), here

ISSN: 2005-4246 IJUNESST
Copyright ⓒ 2015 SERSC

the information can be categorized naturally according to city. China has about three thousand cities, each city is a class. Sometimes users are interested in accident information of some cities, not all. The following is this kind of query

*select count(*) from traffic_accident where city in ('gz','bj','sh') and date between '2014-5-1' and '2014-5-31' group by city having count(*)>100*

The selected city set can be any subset of all three thousand cities. This query has a very important significance, which can shows when and where is accident-prone, and conceptually equivalent to running multiple traditional range queries at the same time.

There is a lot of similar example in real world, for example in Chinese stock database data may be stored as (date, stock, turnover), the stock analysts are concerned how many stocks whose turnover is beyond some specific value in specific date.

According to the above discussion, we can define the kind of problem formally.

Here are some symbol definitions, D is a set which contain N data elements, each element has a key which belong to real domain, and each element belongs to one and only one class. Let K be total class number, $D_k$ ($1 \le k \le K$) denotes the dataset whose elements' class is k.

**Definition Multi-class Range Count (MCRC)**

Given an range r in real domain R and a not empty set $P \subseteq \{1,2,3,\ldots,K\}$, for each $k \in P$, MCRC query returns the number of all data elements in $D_k$ whose key value belongs to r. The set P can be any not empty subset of {1,2,3,…,K}.

When the dataset D has only one class, MCRC query becomes traditional range count problem, which has been solved by SB-tree [7].

## 3. Preliminary Solution

SB-tree can solve traditional range count problem elegantly, it can be used to solve MCRC query too [11-15]. First we briefly explain SB-tree, figure1 is a SB-tree, which is a traditional B-tree in essence, there is a counter in each non-leaf node which records the number of data elements in each subtree. R(u) of each node means an interval in real domain, any value in R(u) can fit in the subtree of node u. For example $R(u_2)=(-\infty, 15)$, $R(u_3)=[15, 37)$. If node u is not a leaf node, then R(u) is union of all child nodes' range, so $R(u_1)=(-\infty, +\infty)$.

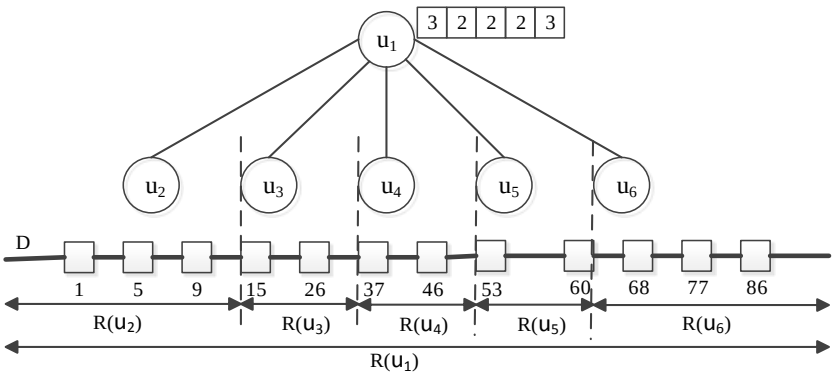



**Figure 1. SB-tree**

In SB-tree, given an range r=[30, 80], range count query starts from root node to leaf node, and visits those child nodes whose range intersects with but not belong to r, the

 Copyright ⓒ 2015 SERSC

number of those child nodes must be 2, in the query, they are $u_3$ and $u_6$. The query visits all child node of $u_3$ and $u_6$ to find the elements whose key belongs to r. All the child node of those node between u3 and u6 must belong to r without being visited, so we can get the counts from $u_1$ directly. Therefore for r=[30, 80], the final query result is 0+2+2+2=6.

We can solve MCRC query by extending SB-tree, Figure 2 is sketch map, non-leaf node $u_1$ has two group counters, each for one class. The count query is similar to the traditional one. This solution that can be named OFA uses one tree to index the data of all classes, each non-leaf node has K counters, the k-th counter records the data amount of the k-th class. Each non-leaf which has O(KC) counters needs O(K) disk pages, each range count query will visit $O(\log_c N)$ non-leaf nodes, so query cost is $O(K\log_c N)$ I/Os, so is update cost.

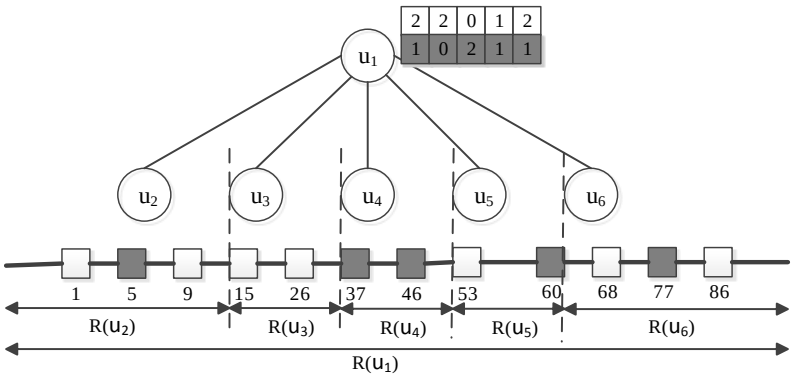



**Figure 2. OFA**

Another intuitive solution is to create a separated tree for each class, namely there are K SB-trees if there are K classes. The solution can be named OFE, as is shown in Figure 3. Let N_max be the most value of element amount among each class, obviously the query cost is $O(|P|\log_c N\_max)$ I/Os, update cost is that of sole B-tree, namely $O(\log_c N\_max)$ I/Os.

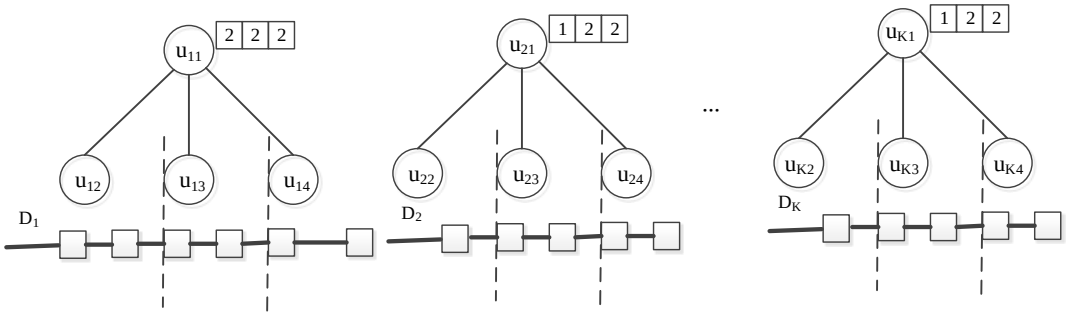



**Figure 3. OFE**

## 4. MCRC-tree

This section we propose final solution to solve MCRC query that can defeat the above drawback, named MCRC-tree, it is actually an ordinary B-tree, which index all data items by key, not like SB-tree, the counter record the number of data items by means of preorder sum. Each non-leaf node has a patch page, namely P(u). The counter and P(u) are stored in continuous space outside the tree node, the tree node only keeps the start address of the counter and P(u).

Figure4 is an example. The example has two classes, white and black, so each non-leaf node has two kind of counters. P(u) stores the information to be updated.

Copyright ⓒ 2015 SERSC

### 4.1. Patch

In MCRC-tree each non-leaf node is attached with a patch page named P(u), in Figure 4, white element 55 is to be deleted, and black element 39 is to be inserted. The two update records are stored in all P(u)s in each non-leaf node on the path from root to leaf that the element belongs to, hence $P(u_1)$ and $P(u_3)$ keep the update information.

We suppose that P(u) can hold three records, then if 30 is to be inserted, P(u) becomes full. If P(u) is full, real update operations need to be triggered, and P(u) will be cleared, we call this procedure counter purge, which is shown as Figure 5. Counter purge needs to adjust corresponding counter value, for example, $R(u_2)=(-\infty, 37)$, $R(u_3)=[37, 68)$, $R(u_4)=[68, +\infty)$, so the second and third value of white counter in $u_1$ corresponding to $u_3$ needs to decrease one

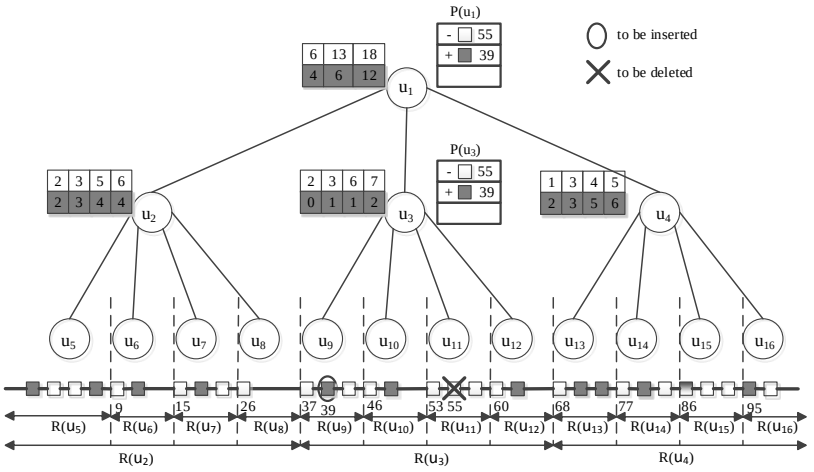
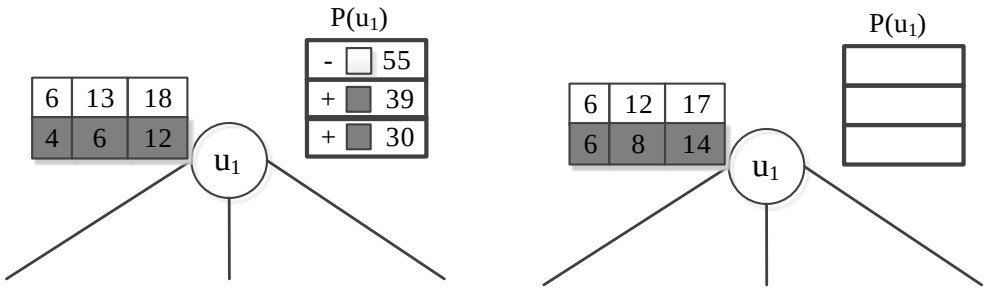



**Figure 4. MCRC-tree**

because of deleting 55. The same logic happens because of inserting 39 and 30, the final result is shown in Figure 5b.




a) before counter purge b) after counter purge

**Figure 5. Counter Purge**

One counter purge needs to visit node u and its P(u), which will read O(K) counter page of u once at most, therefore counter purge can be done in O(K) I/Os.

### 4.2. Query

Now we know P(u) is used to delayed update which can reduce update complexity. Only when P(u) becomes full, the real update operation is triggered. Through the delayed update technology, the update cost can become O(1) I/O. However this may cause the

 Copyright ⓒ 2015 SERSC

counter value is not accurate some time, the counter value is need to be adjusted according to P(u).

We elaborate multi-class range count query procedure, let r=[9, 77], P={1, 2}, class 1(2) corresponds to white(black) elements. The query algorithm starts from root node $u_1$, and find its left and right boundary nodes whose range contains boundary value of r. $R(u_2)=(-\infty, 37)$, $R(u_3)=[37, 68)$, $R(u_4)=[68, +\infty)$, obviously $u_2(u_3)$ is left(right) boundary node. If node u has not left(right) boundary node we define it as adjacent left(right)cousin of u's most left(right) child. Without visiting middle node $u_3$ we can know how many qualified elements in subtree $u_3$ by counter value corresponding to adjacent left cousin of right boundary node minus counter value corresponding to left boundary node, here is (13,6)-(6,4)=(7,2), but the value needs to be adjusted according to $P(u_1)$, the first update record means white counter value minus 1, the second means plus 1, so the result is (6, 3). The similar logic goes on in $u_2$ and $u_4$ until the non-leaf node of the bottom. At last in leaf-nodes sequential scan is needed to get right value.

The MCRC query algorithm only needs to visit two paths from root to leaf, and each visit needs O(1) I/O at most to get counter page, so the query cost is $O(\log_c N)$ I/Os

The MCRC query algorithm derives from SB-tree query algorithm, the query pseudo code is shown below.

```
// given a range r and class set Q, return qualified count for each class in Q
mcrc_query (u, result) { //u is tree node, result is an array to store query result
    if (u is not leaf node){
        u1= left boundary node of u;    // R(u1) contains left boundary of r
        u2= right boundary node of u;    // R(u2) contains right boundary of r
        if (exists node between u1 and u2) {
                store counters in u1 to ca[]; //if no boundary node then initial counter value is 0
                store counters in u2 to cb[];
                adjust ca and cb according to P(u);
                for (each class i in Q) {
                        diff = cb[i]-ca[i];
                        result[i] += diff;
                }
    } else {
        for (each element v in u) {
                if (key of v ∈r) and (class (v) ∈Q)
                result[class (v)] += 1;
        }
        if (u1 is natural)
        // natural means existing left boundary node, or else u1 is adjacent left cousin of u's most left child
            mcrc_query (u1,result);
        if (u2 is natural) and (u1 != u2)
        // natural means existing right boundary node, or else u2 is adjacent right cousin of u's most right child
            mcrc_query (u2,result);
}
```

### 4.3. Update

In Figure 5 we assume that P(u) can contain 3 elements, if one new update comes, then P(u) becomes full, this will trigger real updates, each update needs to adjust corresponding counters, after real updates P(u) becomes empty. Anyway each trigger needs to visit node u and its P(u), which will read and write O(K) counter pages once at most, so each trigger can be finished in O(K) I/Os. Let P(u) contains O(C) elements, then each update in u only needs O(K/C)=O(1) I/O. Each update needs to visit one path from leaf to root, so update cost is $O(\log_c N)$ I/Os.

The essence of update algorithm is node splitting and merging algorithm, which is similar to that of traditional B-tree, the difference is the maintenance of the counter.

Copyright ⓒ 2015 SERSC

Node u splitting has three steps. First, counter purges are triggered on node u and its parent up forcibly, figure6a shows the status to be splitted, figure6b shows the status after counter purge.

Second step is to split node u just like traditional B-tree, and assign new counter value to new splitted node. we assume $u_3$ is splitted to $u_3$' and $u_3$", as is shown in figure6c, counter value in $u_3$' is from $u_3$ directly, while counter value in $u_3$" is calculated. For example, white counter value in $u_3$" corresponding to $u_{11}$ is 2, this is because white counter value in $u_3$ corresponding to $u_{10}$ is 3, while white counter value in $u_3$ corresponding to $u_{11}$ is 5, the difference between the two counter value is white counter value in $u_3$" corresponding to $u_{11}$, we can calculated other counter values in $u_3$'' according to the same logic.

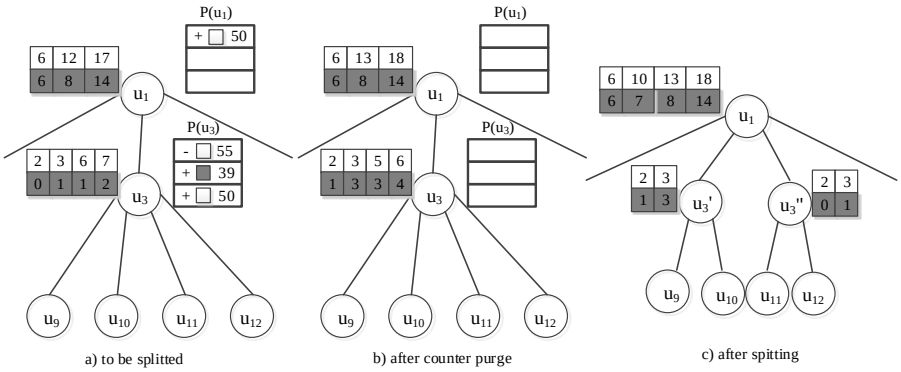



**Figure 6. Node Splitting**

At last the new node u' and u" splitted from u are as the left and right child of up, some useless counter should be deleted and counter in up should be reconstructed. For example, in figure6, after splitting u3 is disappeared, so original counter value in $u_1$ corresponding to $u_3$ should be deleted. The new node $u_3$' is as left child of $u_1$, and $u_3$" as right child of $u_1$. New counter in $u_1$ corresponding to $u_3$' and $u_3$" need to be created, the counter value in $u_1$ corresponding to $u_3$" is copied from old counter value in $u_1$ corresponding to $u_3$, while the counter value in $u_1$ corresponding to $u_3$' needs to be calculated, whose value is the difference between the counter value in $u_1$ corresponding to $u_3$" and the counter value in $u_3$" corresponding to its most right child $u_{12}$.

One node splitting needs two counter purge at most, and read/write counter page of four nodes at most, so splitting cost is O(K) I/Os.

The splitting algorithm pseudo code is as follows.

```
mcrc _split (u) {
   up = parent node of u;
   perform counter purge on u and up forcibly;
   split u into left node ul and right node ur as traditional B-tree;
   for (each child node uc of u')
      counter values in ul corresponding to uc = corresponding counter values in u;
   for (each child node uc2 of u'')
      counter values in ur corresponding to uc2= counter values in u corresponding to uc2 –
counter values in ul corresponding to its most right child;
   delete node up;
   set ul and ur as left child and right child of up;
   counter values in up corresponding to ur = original counter values corresponding to u;
   counter values in up corresponding to ul = counter values in up corresponding to ur – counter
values in ur corresponding to its most right child;
}
```

 Copyright ⓒ 2015 SERSC

The merging algorithm is inverse process of splitting algorithm, merging cost is the same as splitting cost, the pseudo code is as follows.

```
mcrc _merge (ul, ur) {
      up = parent node of ul;
      perform counter purge on up, ul and ur forcibly;
      merge ul and ur into u as traditional B-tree ;
      for (each child uc of node u){
         if(uc comes from ul){
            counter values in u corresponding to uc = original counter values in ul corresponding
to uc;
         }else{
            counter values in u corresponding to uc = counter values in ur corresponding to uc -
counter values in ul corresponding to its most right child;
         }
      }
      delete node ul and ur;
      set node u as child node of up;
      counter values in up corresponding to u = original counter values in up corresponding to
ur;
}
```

## 5. Experiments

This section we will confirm the performance of MCRC-tree through enough experiments, the data that we use in experiments are uniformly distributed in the interval $[0, 2^{40})$. Because the performance of OFE is better than that of OFA in theory, we will compare MCRC-tree to OFE in experiments.

The first experiment assesses how the query performance is affected by the size of class set. The total number of data is 8 million, and the total number of class is 800, the size of the query class set P varies in interval [1, 10], figure7 shows the relationship between query cost and |P|. We can see that the query cost of MCRC-tree remains almost unchanged no matter how big the |P| is, that's because the relationship between query cost and |P| is logarithmic. With the increasing of |P| the query cost of OFE becomes larger, it is approximate linear relationship between query cost of OFE and |P|. Also we can see an important conclusion that the performance of OFE is better than that of MCRC-tree when |P| is small enough, in the figure7 is about 6.

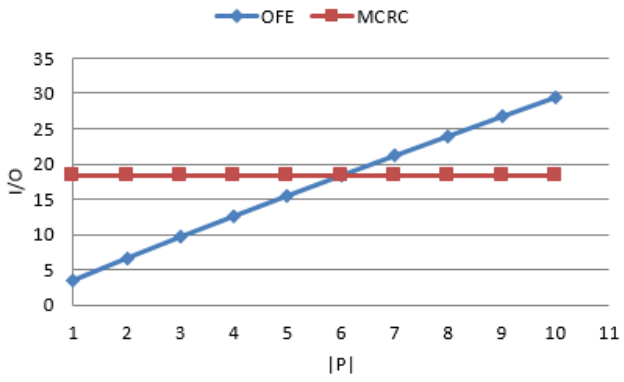



**Figure 7. Query Cost and |P|**

The second experiment assesses how the query performance is affected by the data amount N, the data amount N varies from 8 million to 64 million, other parameters are fixed, we can see from figure8 that the query cost has tiny change when the data amount

Copyright ⓒ 2015 SERSC

N changes, the result is reasonable because the query cost and the data amount is logarithmic relationship in theory.

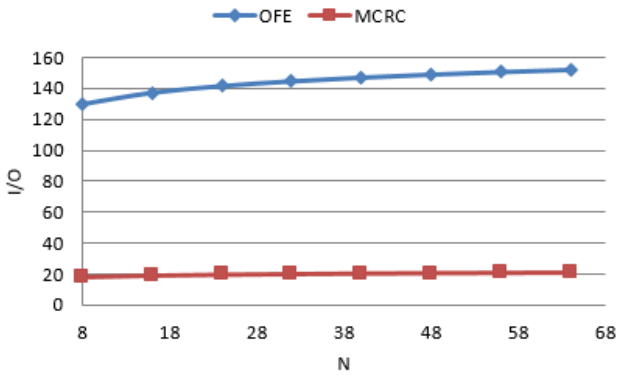



**Figure 8. Query Cost and N**

The third experiment studies update cost, we define ρ as the ratio of inserts to deletes in update sequence, the element to be inserted is uniformly distributed in the data domain R, while the element to be deleted is randomly selected from dataset D. In the experiment the data amount N is 4 million, the class amount is 400. Figure9 shows the relationship between the update cost and ρ, from the Figure 9 we can see that the update cost remain stable no matter how ρ changes. In theory the update complexity of both OFE and MCRC-tree is $O(\log_c N)$ I/Os, which matches the experimental result. Also we can see that the performance of OFE is better than that of MCRC-tree, that is because MCRC-tree needs to maintain counter value and patch.

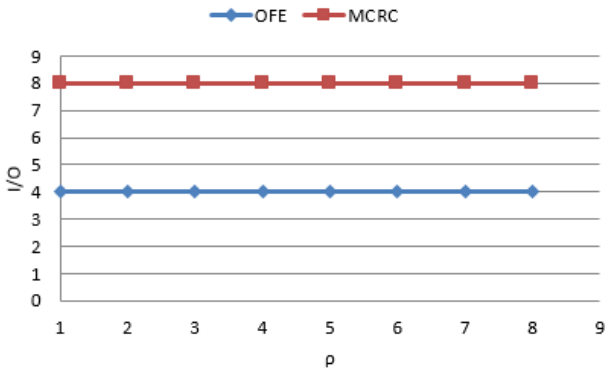



**Figure 9. Update Cost and ρ**

The fourth experiment studies space cost, let K be 400, Figure 10 shows the relationship between the space cost and the data amount N is linear, while MCRC-tree needs more space than OFE, that is because MCRC-tree needs extra space to store counter value and update patch.

On the one hand from the above four experiments we can see that OFE has serious flaw. To be specific, when the length of the MCRC query set P is quite large, the query cost of OFE is too expensive. However we know that supporting large-scale length of the MCRC query is exactly significant in practice; on the other hand the MCRC-tree has very good performance in query, update and space requirement, which is worth adopted in practical application.


Copyright ⓒ 2015 SERSC

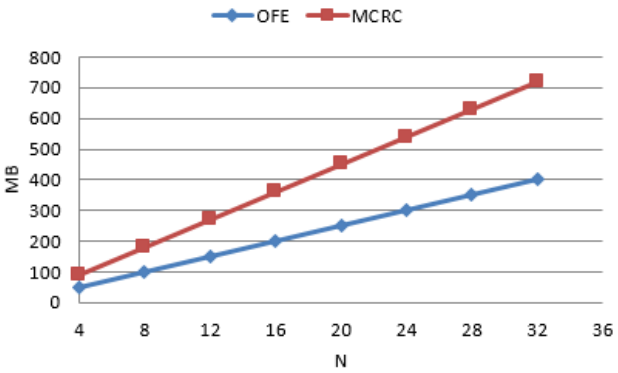



**Figure 10. Space Cost and N**

## 6. Conclusions and Future Work

Multi-class range count query is extension of traditional range count query, which is equal to running many range count query on a large quantity of classes on the same time. It is a challenge to solve the problem in highly-efficient way. This paper proposed a new data structure based on B-tree named MCRC-tree, which can solve multi-class range count query efficiently, especially the query set P can be any non-empty subset of query class set, under any query parameters the query cost is still $O(\log_c N)$ I/Os, so is update cost, C is the disk page size, N is the total data amount. The abundant experiments prove the capability of MCRC-tree.

Our future direction is to extract some performance theorems and testify them in mathematical way, also we try to find some better way to solve multi-class range count query.

## Acknowledgments

The authors wish to thank the National Natural Science Foundation of China for contract 61070015, the natural science foundation of Guangdong province of China for contract 10351806001000000, the fundamental application research project in Guangzhou of Guangdong province of China for contract 11C41150785, under which the present work was possible.

## References

**[1]** D. J. Abadi, A. Marcus, S. R. Madden adn K. Hollenbach, "SW-Store: a vertically partitioned DBMS for Semantic Web data management", The VLDB Journal, vol. 2, (**2009).**

[2] A.-S. Charest, "Empirical evaluation of statistical inference from differentially-private contingency tables", In: Proceedings of Privacy in Statistical Databases-PSD 2012, LNCS 7556, Springer (**2012**), pp. 257–272.

[3] J. Dittrich and J.-A. Quiané-Ruiz, "Efficient parallel data processing in MapReduce workflows", PVLDB vol. 5, (**2012**), 2014–2015.

[4] F. Halim, S. Idreos, P. Karras and R. H. C. Yap, "Stochastic database cracking: towards robust adaptive indexing in main-memory column-stores", PVLDB, vol. 5, no. 6, (**2012**), pp. 502–513.

[5] H.-C. Yang and D. S. Parker, "Traverse: simplified indexing on large Map-Reduce-merge clusters", In: DASFAA, (**2009**), pp. 308–322.

[6] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden and I. Stoica, "Blink and it's done: interactive queries on very large data", Proc. VLDB Endow., vol. 5, no. 12, (**2012**), pp. 1902–1905.

[7] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates", VLDB J., vol. 12, no. 3, (**2003**), pp. 262–283.

[8] N. Sarkas, G. Das, N. Koudas and A. K. H. Tung, "Categorical skylines for streaming data", In SIGMOD, (**2008**), pp. 239–250.

[9] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger, "On computing temporal aggregates with range predicates. TODS, vol. 33, no. 2, (**2008**).

Copyright ⓒ 2015 SERSC

[10] C. Qin and F. Rusu, "PF-OLA: a high-performance framework for parallel online aggregation", Distributed and Parallel Databases, (**2014**), p. 32.
[11] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multi-version B-Tree", VLDB Journal, vol. 5, no. 4, (**1996**), pp. 264–275.
[12] B. Chazelle, "A Functional Approach to Data Structures and Its Use in Multidimensional Searching", SIAM Journal on Computing, vol. 17, (**1998**), pp. 427–462.
[13] D. Gao, J. A. G. Gendrano, B. Moon, R. T. Snodgrass and M. Park, B. C. Huang and J. M. Rodrigue, "Main Memory-Based Algorithms for Efficient Parallel Aggregation for Temporal Databases", Distributed and Parallel Databases, vol. 16, no. 2, (**2004**), pp. 123–163.
[14] B. Salzberg and V. J. Tsotras, "Comparison of Access Methods for Time-Evolving Data", ACM Computing Surveys, vol. 31, no. 2, (**1999**), pp. 158–221.
[15] Y. Tao and D. Papadias, "Range Aggregate Processing in Spatial Databases", In IEEE Transactions on Knowledge and Data Engineering (TKDE), (**2004**), pp. 1555–1570.

## Authors

**Hu Jinglin**, he is a candidate of PhD in South China University of Technology, also he is a lecturer in Nanchang HangKong University. His current research interests include algorithm, database and big data.

**Qi Deyu**, he is a professor in South China University of Technology, and director of Research Institute of Computer Systems at South China University of Technology. His current research interests include distributed systems, computer architecture, network security and software architecture.

 Copyright ⓒ 2015 SERSC