ATLANTIS PRESS

Atlantis Highlights in Engineering (AHE), volume 2

International Conference on Industrial Enterprise and System Engineering (IcoIESE 2018)

Cost-Based Storage of the R-tree Aggregated Values over Flash Memory

Wojciech Macyna* and Krzysztof Majcher†
*Institute of Computer Science
Wrocław University of Technology, Poland
Email: wojciech.macyna@pwr.edu.pl
†Institute of Computer Science
Wrocław University of Technology, Poland
Email: k.majcher@pwr.edu.pl

***Abstract*—The flash memory due to its shock - resistance, power economy and non-volatile nature is considered as a very popular storage device. It is widely used in mobile phones, sensor networks and hand-held devices. What attracts the attention is the data storage in the flash memory. Database vendors try to adapt their systems to the flash memory architecture. To make it efficient, database indexes, such as: B-tree or R-tree, must be implemented from the scratch in order to take into account the flash memory limitations. The R-tree is wildly used in the geographical databases for indexing the spatial objects. The aggregated R-tree (aR-tree) extends the original R-tree with the aggregated values, which are connected with the nodes.**

In this paper, we deal with the materialization technique of the aggregated values in the aR-tree index. We claim that not all the aggregated values should be explicitly stored in the flash memory. We propose a Cost-Based Method which chooses the most beneficial set of the aggregated values for materialization. Our method is particularly beneficial for devices with limited capacity and is optimized for the flash memory architecture.

***Index Terms*—flash memory, R-tree, spatial database**

I. Introduction

A quick access to information is a key point for the success for commercial organisations and domestic enterprises in the competitive global market. The overwhelming data amount requires effective tools for selecting, interpreting and deriving the useful information from them. Nowadays, the popularity of mobile devices with limited memory and energy supply still grows. In such devices, the flash memory is applied. This is because of its shock - resistance, power consumption and non-volatile nature. The flash memory is particularly utilized in: sensor networks, embedded systems or hand-held devices. Apart from that the solid state disks which are build upon flash memory chips, become popular in personal computers.

Flash disks have different characteristics in comparison to traditional hard disks. A flash memory block consists of the fixed number of pages (32 or 64). The read and write operations are performed on the page granularity whereas in order to delete the data, the entire block must be erased. Moreover, the write operation takes more time and energy than the read one. Such limitations must be considered in the implementation of almost all data storage techniques.

The growing popularity of the spatial and geographical databases requires efficient storage methods. The best solution to speed up the access to the required area in the spatial database is to use the R-tree index. This structure was proposed in [1]. Since then, many variants of this index have been developed. The R-tree index divides the spatial area into rectangle-shaped regions. Every entry in the R-tree corresponds to the rectangle of the considered area and has pointers to the smaller rectangles inside it. The aR-tree index may be treated as an extension of the standard R-tree [2]. In that index, the aggregated values connected with the R-tree node are explicitly stored. Consequently, the aggregated value does not need to be processed every time during query execution, but may be fetched directly from the memory.

As it was mentioned before, the standard implementation of database indexes cannot be applied directly to the flash memory. Therefore, several new types of the B-tree and R-tree have been proposed in: [3], [4] and [5]. The effective storage of aggregated values for the R-tree is presented in [6]. In this paper, the aggregated values are packed into flash memory pages. However, the authors do not consider a situation when the storage space for aggregated values is limited and, consequently, not all aggregated values may be stored. The goal of our work is to fix this deficiency. We claim that storing all aggregated values of the R-tree in the flash memory may be impossible due to the limited memory capacity or excessive storing costs. On the other hand, maintaining none of them determines calculating them every time they are required. It can lead to the computational overhead, time delay and lost of energy. Thus, the balance between materialization and computation of the aggregated values must be found. Moreover, when the aggregated value is not queried or its storage is not beneficial for the system, it should be deleted and replaced by a more profitable one.

Many interesting page replacement methods for flash memory have been proposed: [7], [8], [9], [11], [12]. However, these techniques do not consider a situation where the calculation cost of the data in the tree node N depends on the calculation cost of the data in the child nodes of N.

In this paper, we propose a Cost-Based Method (CBM). The method stores the most beneficial subset of the aggregated values for the aR-tree index considering the flash memory limitations.

The paper is organized as follows. In the next section, we formulate the problem. In section III, we describe an intuitive Cost-Based Method for materialization of the aggregated values in the R-tree. The method utilizes the Aggregation Pool as a space where the aggregated values are stored. Due to the limited size of the Aggregation Pool, it can maintain only the most beneficial aggregated values. The **main contribution** of the paper is presented in section IV. It can be pointed out as follows:

- We adapt the Cost-Based Method to the flash memory.
- We propose a flash aware storage model for our method.
- We carry out several experiments, in which we show that the proposed method is effective for the flash memory and outperforms other techniques (section V).

II. Problem formulation

The main purpose of the R-tree index is to manage the spatial data efficiently. Figure 2 shows an example of the searching area R which consists of three subareas: A, B, C. These subareas are divided into smaller rectangles. The smallest rectangles contain the spatial objects, which are depicted as o_i. Consequently, the R-tree facilitates the searching operation. For example, to find all the objects residing in the rectangle $B1$, the R-tree node containing this rectangle must be found. Figure 1 shows the example of the R-tree index for the spatial area presented in Figure 2. The R-tree node consists of entries. Each entry has the following components: $dataptr$, $childnode$, id, $minimalboundingbox$, which mean: a pointer to data, a pointer to the child node, a pointer to the R-tree node and a minimal bounding box, respectively.

The aR-tree can be considered as the extension of the R-tree ([2] and [6]). It connects the aggregated values with the node of the R-tree. Worth noting is the fact that the computing of the aggregated value may be complex and should not be repeated every time the value is required.

That is why it is better to evaluate them once and then store explicitly in the memory. For example, we can calculate the number of the objects in the rectangle A based on the number of objects in $A1$, $A2$, $A3$ and then materialize this value inside the R-tree. In this way, we could avoid the subsequent calculation of the same value if it is queried in the future. Clearly, there may be many aggregated values assigned to one aR-tree area. Let's suppose a situation that a city map containing many different spatial objects is indexed by the R-tree. The objects may be categorized. Each object may have many attributes, etc. For example, to calculate the average height of the buildings in the area A, only the objects belonged to the category $building$ must be considered. In this sense, there may be many different aggregated values to calculate for each R-tree node. It would be problematic to materialize all of them, particularly in the system with limited memory capacity. On the other hand, the calculation of the aggregated value every time would increase the execution time and energy cost. Therefore, the balance between materialization and calculation is needed.

In many cases, the materialization cost may be much higher than the calculation cost. Let's consider the aggregated function $count$ which calculates the number of objects in the particular area. Let $c(R)$ denote the value of the function $count$ for the node R. Clearly, if the values: $c(A)$, $c(B)$ and $c(C)$ are stored in the memory, it does not pay to store $c(R)$, because it can be easily calculated from their child nodes (Figure 1). All other distributive functions like: sum, max and min can be calculated in the same way, i.e. using the materialized results from the child nodes [2]. Similarly, the algebraic functions, for example, $average$ can be obtained as $sum/count$. Obviously, its value does not need to be materialized, if the sum and $count$ are stored.

III. Proposed implementation

In this section, we provide the Cost-Based Method (CBM) for storing the aggregated values of the R-tree.

A. Materialization algorithm

The materialization algorithm uses two components: the Aggregation Pool and the main storage. The first component is intended to store the aggregated values whereas the second one maintains other data, e.g. the data of the R-tree and other spatial objects. Both components may utilize different memory types. However, in the subsequent parts of this work, we will focus on the flash memory as a storage type. Thus, the Aggregation Pool may be treated as a flash memory cache for storing aggregated values of the R-tree nodes. Apart from that, we utilize a small RAM buffer where we hold the calculated aggregated values. The buffer works following the LRU (Least Recently Used) policy. It means that when the aggregated value is found in the RAM buffer, it is shuffled to the beginning of this buffer. In this way, we favor the aggregated values which are queried frequently.

The algorithm works as follows. When the aggregated value connected with the node N is queried, the system looks for this value in the RAM buffer, then in the Aggregation Pool. If the required value does not exist there, its calculation must be recursively done on the basis of the child nodes of N.

When the calculation of the aggregated value related to the node N is completed, an aggregated item I_N is created. It is composed of: Id, val and $cost$, which denote: a node identifier, a set of aggregated values and a calculation cost of the aggregated value, respectively. The aggregated item may be stored in the Aggregation Pool, if it is profitable for the system.

In Figures 3 and 4, we visualize the materialization method. In this example, we use the R-tree, which consists of three levels. The node R is the root node. The middle layer has three nodes: A, B and C. They are pointed by the entries of the root node R. Let us suppose that two aggregated items are stored in the Aggregation Pool: I_A and I_{C_1} (Figure 3) . So, the values related to the nodes A and C_1 do not have to be evaluated. If the request about the value of the root node is sent, the calculation should be done on the basis of the item I_A and the subtrees of B and C. Furthermore, the aggregated

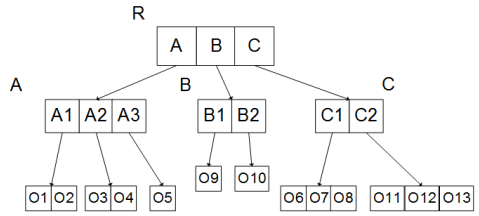

Fig. 1. R-tree for the indexed area

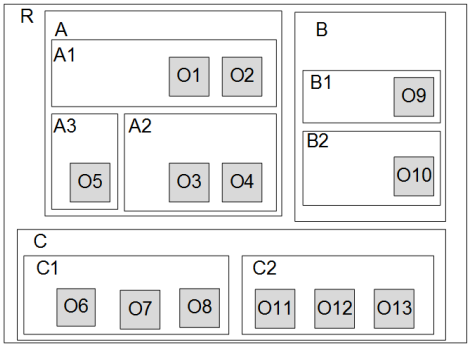

Fig. 2. Indexed area

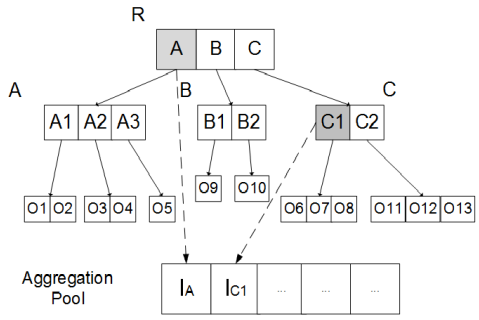

Fig. 3. aR-tree before inserting $O14$

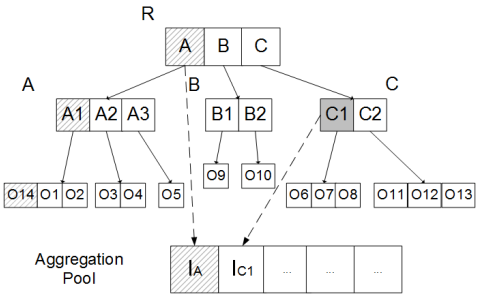

Fig. 4. aR-tree after inserting $O14$

item I_{C_1} may be helpful for calculation of the values related to the node C. As a result, the calculation cost is much smaller than the calculation without any materialization. It comes from the fact that materialized values are stored explicitly and do not have to be calculated from the scratch. Such calculation would impose the additional overhead since it would require traversing from the node down to the leaf nodes.

Clearly, a problem arises when the R-tree or the values of leaf nodes are updated (Figure 4). Let's suppose that the R-tree is modified by adding an object $O14$. As a consequence, the aggregated value related to the node A may change. This implies the changing of I_A what should be reflected in the Aggregation Pool. In our method, we utilize the lazy approach [10]. Instead of recalculation of I_A, we simply remove the invalid item from the Aggregation Pool. Clearly, a new version of I_A may be calculated later, if required.

B. Query evaluation cost

Based on the intuition described above, we introduce a definition of the query cost for our method. The cost is strongly connected with the memory access costs. These factors are particularly useful in the flash memory context.

Let AP denote a set of the aggregated items belonging to the Aggregation Pool. Let r_I, w_I and d_I mean the approximate reading, writing and deleting cost of one item in the Aggregation Pool, respectively. Additionally, r_L is the reading cost of the aggregated value of the R-tree leaf node.

We define the query evaluation cost $c(R)$ for the node R as follows:

$$c(R) = \begin{cases} 0 \text{ if } I_R \in \text{RAM} \\ r_I \text{ if } I_R \in \text{AP} \\ r_L \text{ if } R \in R_{leafnodes} \\ \sum_j c(R_j) \text{ for each } R_j \in R_{children} \end{cases}$$

Let's introduce a notion of the transformation cost. It can be defined as the cost of removing an aggregated item from the Aggregation Pool (d_I) and the cost of adding the other aggregated item into it (w_I):

$$transform(I, I_1) = w_I + d_I \tag{1}$$

The Materialization algorithm (see Algorithm 1) works as follows. If there is sufficient place in the Aggregation Pool, the created aggregated item (I) may be added to it (line 2). Otherwise, the cost of this item is compared with the smallest cost of the aggregated item already stored in the Aggregation Pool (I_1). If the condition in line 5 is fulfilled, the aggregated item I_1 is replaced by I. Thus, the aggregated item is materialized only in a situation, if it is profitable for the system.

IV. Adapting to the Flash Memory

In this section, we consider the adaptation of CBM to the flash memory. We focus on the flash memory as a storage type for the Aggregation Pool.

Algorithm 1 Materialization(input I)

Require: Item I
Require: Item set in the Aggregation Pool: AP
1: **if** $AggregationPool\ is\ not\ full$ **then**
2: $AP \longleftarrow AP + I$
3: **else**
4: $I_1 \longleftarrow GetWorstInAggregationPool()$
5: **if** $cost(I) > cost(I_1) + transform(I, I_1)$ **then**
6: $AP \longleftarrow AP - I_1$
7: $AP \longleftarrow AP + I$
8: **end if**
9: **end if**

Algorithm 2 Merge()

Require: Set of pointers in the mapping table: S
Require: Set of pages in U-blocks: U
Require: Set of pages in D-blocks: D
Require: Set of pages in D-blocks to reclaim: D_{rec}
Require: The maximal number of items in one page: t
1: Sort the aggregated items in the U-blocks by Id
2: Identify a set of D-pages to reclaim: D_{rec}
3: **for all** $s_i \in S$ **do**
4: **if** s_i points to the page $d_i \in D_{rec}$ **then**
5: Merge items from d_i with such items from u_i which are within the range of d_i
6: **if** $|d_i| + |u_i| \leq t$ **then**
7: Create a new page d_i' with the items from d_i and u_i
8: Replace s_i by s_i' and set its pointer to d_i'
9: **else**
10: Create a new subset D_i" of pages: $|D_i"| = \lceil \frac{|d_i|+|u_i|}{t} \rceil$
11: Allocate items from d_i and u_i to D_i"
12: Modify the range of pages in D_i"
13: Modify the mapping table by removing s_i and adding pointers S_i" to the new pages
14: **end if**
15: **end if**
16: **end for**

To efficiently store the data in the flash memory, we propose the flash translation layer, which utilizes two kinds of memory blocks: D-blocks and U-blocks [13]. They consists of D-pages and U-pages, respectively (see Figure 5). The D-blocks store the data and the U-blocks maintain modifications of data. Apart from that, we use the mapping table, which is stored in RAM. The item of the mapping table holds the pointer to the page P_i in the D-block, the identifiers range of the stored aggregated items and the minimal item cost c_i inside the page P_i. In this way, we can keep the aggregated items sorted, for example, by their identifiers. As a consequence, the system can easily identify the presence of the requested item in the Aggregation Pool. To optimize the flash memory utilization, we introduce a notion of the deleted aggregated item I' which holds the identifier of I. The presence of I' in an U-block means that the aggregated item I must be deleted during the merge process.

The operations on data are defined as follows.
Insertion. The inserted aggregated item I is stored in the page of the U-block. If the Aggregation Pool is full, the deleted version of the "worst" aggregated item from the D-blocks is stored in the U-block as well.
Deletion. When the item I must be removed, its deleted version I' is inserted into the U-block.
Update. The operation is treated as a sequence: $deletion$ and $insertion$. So, if the aggregated item I is updated, the system deletes I by inserting I' and then inserts the new version of I.

If there is no space in the U-blocks, the $Merge$ algorithm (see Algorithm 2) is invoked. It works as follows. First, it sorts the aggregated items in the U-pages by Id. If some aggregated items from the U-pages are within the range of a D-page, they are fetched and packed to the new allocated D-pages. The number of new D-pages depends on the number of the aggregated items (line 10). Then, the mapping table is updated. The pointers to the new D-pages and the new ranges connected with these pointers are stored in the mapping table. Clearly, the old pages are denoted as obsolete and are left for the Garbage Collector to reclaim.

In Figures 5 and 6 the memory reclamation process for CBM is presented. In this example, the available memory consists of three pages in D-blocks and one page in an U-block.

Before the merge (Figure 5), two operations are performed. The first operation is an update of the aggregated item $I5$ which causes insertions of two items into the U-block: $I5'$ and a new version of $I5$. The second one is insertion of $I19$. As it was described above, in order to insert a new item, the "worst" item should be deleted. Let's suppose that c_1 is the cost of $I4$ and c_1 is the worst item cost in the Aggregation Pool. In that situation $I4$ must be removed. This is performed by inserting $I4'$ into the U-block.

When the U-block is full (Figure 5), the merge process is invoked. The aggregated items: $I5$, $I5'$ and $I4$ from $U1$ are merged with the aggregated items in $P1$, because their identifiers are within the range of $P1$. As a result, a new page $P1'$ is created. The aggregated item $I19$ is merged with the items from $P3$. It creates two new pages: $P3'$ and $P4'$. Please note that the page $P2$ remains intact. After that the merge process causes an update in the mapping table. Now, it contains 4 entries with the new ranges. Finally, the U-blocks are erased and ready for the further use (Figure 6).

V. Performance evaluation

In this section, we shall discuss some simulations which confirm the effectiveness of the proposed method. For the experiments, we used the flash with the following characteristic. The NAND flash device is organized as a collection of blocks, with 64 pages per block. The page size is 2048 bytes. The page read time is 130.9μs, the page write requires 405μs and the

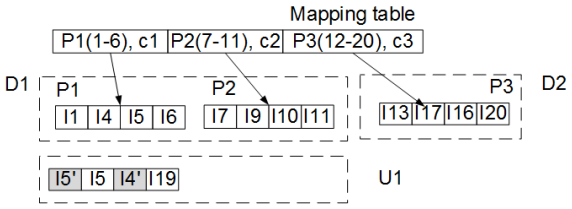

Fig. 5. CBM: State before the reclamation process

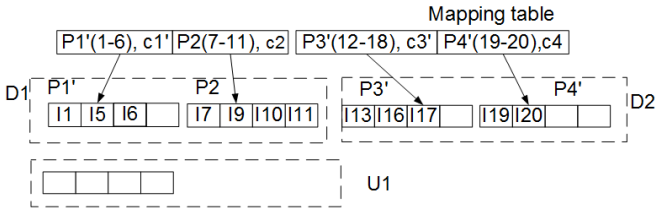

Fig. 6. CBM: State after the reclamation process

block erasing time is $2000\mu s$. We created the R-tree, which consists of 5 levels and contains 100000 leaf nodes. Every single inner node has between 10 and 20 child nodes. We claim that the R-tree with such parameters is typical in the real applications. For implementation simplicity, we assume that each R-tree node is stored in one flash page. We use the Aggregation Pool of the fixed size. The size denotes the maximal number of aggregated items which can be stored in the Aggregation Pool. In the experiments, we utilize the RAM memory buffer, where the aggregated items may be stored. In the RAM buffer we follow the Least Recently Used (LRU) policy. In this way, we favor the aggregated values which are asked frequently.

In our experiments, we use the following query patterns:

1) *High level pattern.* The aggregated values from the highest levels of the R-tree are queried.
2) *Middle level pattern.* The aggregated values from the middle levels of the R-tree are queried.
3) *Low level pattern.* The aggregated values from low levels of the R-tree are queried.
4) *Random distribution.* The aggregated values from all R-tree nodes are queried with the same probability.

Besides, we consider two different workloads:

1) *Read intensive workload.* The read operations are dominant in this workload. One write operation is performed every 10 read operations.
2) *Write intensive workload.* The write operations are dominant in this workload. One read operation is performed every 10 write operations.

Please note that the changing of the aggregated values in the leaf node implies the recursive changing of the aggregated values of all nodes at the levels above. As a consequence, all not valid aggregated values must be deleted from the Aggregation Pool.

In the first experiment (Figure 7), we present the elapsed time depending on the Aggregation Pool size. For the read intensive workload, we performed 10000 read operations and 1000 write operations. For the write intensive workload, we carried out 10000 read operations and 100000 write operations. As we can see, the Aggregation Pool size affects the performance in the case of the read intensive workload, but it does not impact the performance for the write intensive workload. When the Aggregation Pool size is small, the number of stored aggregated values is small as well, what increases drastically the average query response time. On the other hand, the performance of the write intensive workload is spoilt by a great number of modifications in the Aggregation Pool.

In the second experiment (Figure 8), we measured the elapsed time after the requests of the nodes in the different levels of the R-tree. In this case, we use only the read intensive workload. From this experiment, we can conclude that the query evaluation time is better for the nodes in the lower levels of the tree (near the leaves nodes). The result for the higher levels is worse because the whole subtrees must be traversed to calculate the aggregated values. Clearly, the performance is better, when the size of the Aggregation Pool is bigger. It comes from the fact that holding more aggregated items in the Aggregation Pool makes the calculation more effective.

In Figure 9, we present the number of the Aggregation Pool replacement for the different query patterns. The result is very high, when the high levels of the R-tree are queried and the Aggregation Pool size is small. For 100 and 500, the number is equal to 0 since all requested aggregated values are stored in the Aggregation Pool. In the case of the middle levels, the number of reads grows slightly with the increasing size of the Aggregation Pool.

Figure 10 compares the proposed method with the method which does not use the Aggregation Pool. Axis Y shows the gain obtained with CBM. It is calculated as: $(c_n - c_a)/c_n * 100$, where c_n and c_a mean the cost of the method without materialization and the cost of CBM, respectively. As we can see, the gain is very high, when the nodes in the high levels of the R-tree are requested. The gain is much smaller, when the nodes from the middle levels are requested and when the Aggregation Pool size is small.

In Figure 11 we present the performance depending on the R-tree size. The consequence of the growing number of tree leaves is the increase of the number of the tree levels. Thus, the performance depends on both: the R-tree size and the Aggregation Pool size.

In the last experiment we implemented the flash aware FIFO (First In First Out) and compare CBM against it. In FIFO, the pages are added into the memory in the order of their arrival. The recent appeared data are stored at the head of the page list and the latest arrived at the tail. So, FIFO does not index the items by their identifiers. As a consequence, the system must look up the entire memory to find the requested aggregated item. Apart from that, FIFO writes every requested aggregated item to the memory. Thus, the performance is spoilt by the big number of writes on the flash. As we can observe in Figure

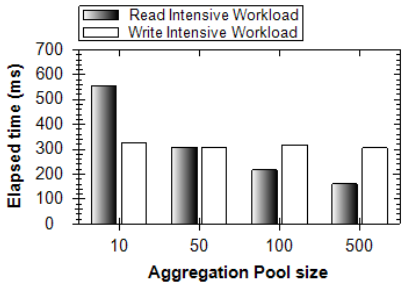

Fig. 7. Performance depending on the workload type

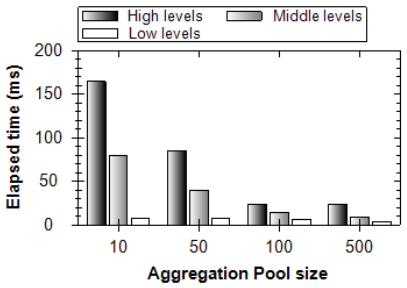

Fig. 8. Performance depending on the requested levels

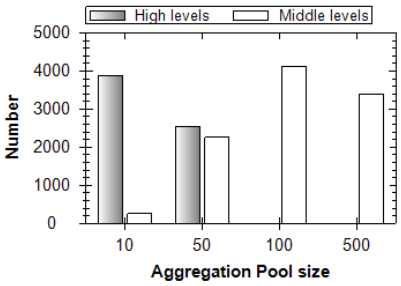

Fig. 9. Aggregation Pool replacement

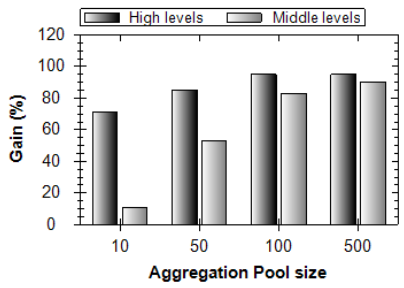

Fig. 10. CBM vs. no materialization

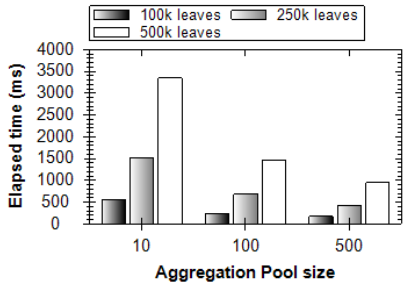

Fig. 11. Performance depending on the R-tree size

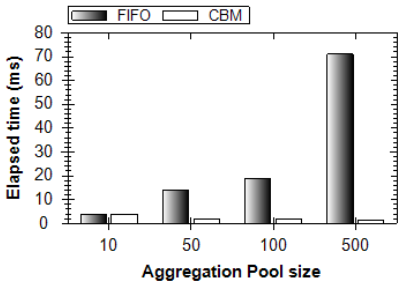

Fig. 12. CBM vs. FIFO

12, CBM drastically outperforms FIFO.

VI. Conclusions and Future Work

In this paper, we propose the Cost-Based Method which enables the materialization of the aggregated values of the R-tree in the flash memory. The method chooses such values which storage is the most beneficial with regard to the computational performance. Our algorithm utilizes the flash memory with limited capacity as a space where the aggregated values are stored. We propose the flash aware storage model, which increases the performance of our approach in the flash memory context. We show that Cost-Based Method outperforms other approaches as far as the flash memory features are concerned.

In the future work, we are going to deal with the materialization of the database views in the flash memory architecture.

Acknowledgment

The paper was supported by Wroclaw University of Science and Technology (grant number 0401/0086/16).

References

[1] Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: International Conference on Management of Data, ACM (1984) 47–57
[2] Papadias, D., Kalnis, P., Zhang, J., Tao, Y.: Efficient olap operations in spatial data warehouses. In: In: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases. (2001) 443–459
[3] Nath, S., Kansal, A.: Flashdb: dynamic self-tuning database for nand flash. In Abdelzaher, T.F., Guibas, L.J., Welsh, M., eds.: IPSN, ACM (2007) 410–419
[4] Wu, C.H., Kuo, T.W., Chang, L.P.: An efficient b-tree layer implementation for flash-memory storage systems. ACM Trans. Embedded Comput. Syst. **6**(3) (2007)
[5] Wu, C.H., Chang, L.P., Kuo, T.W.: An efficient r-tree implementation over flash-memory storage systems. In: GIS, ACM (2003) 17–24
[6] Pawlik, M., Macyna, W.: Implementation of the aggregated r-tree over flash memory. In: Proceedings of the 17th international conference on Database Systems for Advanced Applications. DASFAA'12, Berlin, Heidelberg, Springer-Verlag (2012) 65–72.
[7] Lee, S., Bahn, H., Noh, S.H.: Clock-dwf: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures. IEEE Transactions on Computers **63**(9) (2014) 2187–2200
[8] Fan, Z., Du, D.H.C., Voigt, D.: H-arc: A non-volatile memory based cache policy for solid state drives. In: MSST, IEEE Computer Society (2014) 1–11
[9] On, S.T., Gao, S., He, B., Wu, M., Luo, Q., Xu, J.: Fd-buffer: A cost-based adaptive buffer replacement algorithm for flashmemory devices. IEEE Transactions on Computers **63**(9) (2014) 2288–2301
[10] Zhou, J., Larson, P.A., Elmongui, H.G.: Lazy maintenance of materialized views. In: Proceedings of the 33rd International Conference on Very Large Data Bases. VLDB '07, VLDB Endowment (2007) 231–242.
[11] Swain, D., Dash, B.N., Shamkuwar, D.O., Swain, D.: Article: Analysis and predictability of page replacement techniques towards optimized performance. IJCA Proceedings on International Conference on Recent Trends in Information Technology and Computer Science (2012) 12–16 Full text available.
[12] Park, S.y., Jung, D., Kang, J.u., Kim, J.s., Lee, J.: Cflru: A replacement algorithm for flash memory. In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems. CASES '06, New York, NY, USA, ACM (2006) 234–241.
[13] Kang, J., Jo, H., soo Kim, J., Lee, J.: A superblock-based flash translation layer for nand flash memory. In: In EMSOFT 06: Proceedings of the 6th ACM and IEEE International conference on Embedded software, ACM (2006) 161–170