ATLANTIS PRESS

*Atlantis Highlights in Engineering (AHE), volume 2*

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

# *Flash-aware Clustered Index for Mobile Databases*

Wojciech Macyna* and Michal Kukowski†
*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: michalkukowski10@gmail.com

***Abstract*—Flash memory become the very popular storage technology. Recently, it may be treated as a main storage memory in mobile devices, PDA and personal computers. However, the architecture based on flash memory has some limitations. They suffer from poor write performance, as the flash blocks must be erased before write. In particular, small random writes are very inefficient in comparison to read operations. Such asymmetry has implications as far as data management is concerned. Many database storage techniques must be changed to consider the new hardware characteristics. This paper proposes a new clustered index which considers the flash memory limitations. Due to utilizing fractional cascading and tree-like structure, the efficiency of update and search is obtained. The analytical and experimental results show that the proposed approach outperforms the traditional clustered index based on B+tree in terms of the flash memory limitations. The proposal may be useful in the mobile databases which operate on the flash memory.**

***Index Terms*—flash memory, clustered index, mobile database**

## I. Introduction

The flash memory technology has achieved great popularity among hardware vendors. It is hard to imagine cell phones, sensor networks, music players or PDA devices without flash memory cards. This is due to their still growing capacity, excellent shock resistance, low power consumption and high random read performance.

The flash memory is organised as a set of blocks. Each block contains a fixed number of pages. Read and write operations are performed on the page granularity. However, bits can only be cleared by erasing the whole block. A distinguishing feature of the flash memory is the asymmetry of the read and write speed. The write operation is significantly slower than the read one. Furthermore, an erase is an order of magnitude more time consuming than write. The serious drawback of the flash memory is the limited number of the erase operations per block (about 100000 times). After that, the block may be worn out and no more usable. To solve the above problems, the flash translation layer (FTL) has been proposed. This is an abstraction level which maps the logical memory addresses to the physical ones using a mapping table. It supports a not-in-place update and provides the effective algorithm for leveling the wear of the memory blocks.

There are two categories of FTL schemas: page-level FTL and block-level FTL. In the first category, the mapping is performed on the page granularity. In this case, when the data in the page changes, the new version of the data is written to the other page and the mapping in the FTL is updated. This technique is very flexible but its disadvantage is the growing mapping table size. On the other hand, in the block-level mapping, when the data in the page changes, all unchanged pages are moved to the other block and after that, the mapping table is modified. The advantage of this approach is the relatively small size of the mapping table, since the number of blocks is much smaller than the number of pages. The FTLs are described in more detail in: [1], [2], [3], [4].

Recently, many different mobile database servers have been developed. Many of them like: SQL lite, Microsoft SQL Server Compact and Oracle lite have achieved the commercial success ( [5], [6]). In principle, the popular database management systems support two kinds of indexes: a clustered and a non-clustered index. Both are represented as a B+-tree structure. The clustered index is a special type of index that holds the records of the table physically ordered. The leaf nodes of a clustered index contain the data pages. On the contrary, the leaf node of a nonclustered index does not consist of the data pages. Instead, the leaf node holds the reference to the clustered index.

An insertion of the record into the clustered index is carried out as follows. First, the clustered index must be scanned to find the right place for insertion. Then, the record must be written to the memory. The index may be reorganized if needed. As long as it is not a problem for a traditional hard disk, the flash memory may be exposed to a huge number of write and erase operations. It may happen in the case of mobile databases which operate on small devices equipped with the flash memory chips.

Several types of the nonclustered flash-aware indexes have been proposed. Many of them, like BFTL [7] are write-optimized, but suffer poor search performance. The others, like FD-tree [8] and LA-tree [9] are read and write optimized. In [10], the authors propose Bloom filter tree index. A leaf node of the index consists of one or more Bloom filters which store the information whether a key for the indexed attribute exists in a particular range of data pages. In [11], the flash translation layer for column oriented databases was proposed.

In this work we propose the flash-aware clustered index (FA-index) which may be useful for the mobile databases. We assume that the database records are stored in a tree-like

structure. In this way, we preserve the logarithmic search time. On the other hand, the database records may be maintained not only in the leaves but also in the intermediate levels of the tree. As a consequence, we avoid the reorganization of the whole tree while inserting. Moreover, using the merging technique and bulk insertion approach, we reduce the number of expensive flash erase operations, since we operate on the limited number of memory blocks.

The contribution of the paper is as follows. We propose a new clustered index for the flash memory which has the good update performance and preserves the search efficiency. We estimate the search and insertion time and conduct several experiments to compare the FA-index with the clustered index based on the traditional B+tree.

## II. FA-INDEX

In this section, we present our approach for storing the flash aware clustered index.

Our index is a structure which consists of multiple levels. The top level ($L_0$) is a small, standard B+-tree called a head tree, which size is equal to the block size of the flash disk. As the head tree is very small, we assume that it is stored in RAM. Other levels ($L_i$) consist of memory blocks. The leaf nodes of the head tree contain pointers to the level $L_1$ and the non-leaf nodes of the level $L_i$ contain pointers (called fences) to the next level $L_{i+1}$.

We differentiate four kinds of entries:

1) *Normal entry.* A normal entry denotes a typical database record with a key value denoted as $key$.
2) *Deleted entry.* A deleted entry $d_{key}$ is stored as a $key$ of the record which must be deleted. Thus, if we want to delete a record with a $key$, a new deleted entry $d_{key}$ must be inserted into the head tree. In this way, the deletion is not performed immediately, but is postponed until the entire index is reorganized (see section II-A3).
3) *External fence.* The external fence contains two elements: a key value, $key$, a $pid$, which is the identifier of the page in the level below. For each page $P$ from $L_{i+1}$, one external fence with $pid$ set to $P$ is created and inserted into $L_i$.
4) *Internal fence.* The internal fence contains two elements: a key value $key$ and $pid$. It is used when the first entry of page $P$ is not an external fence. In that case, we add an internal fence with $key$ equal to the value of the first normal entry in the page $P$. Moreover, its $pid$ is the same as $pid$ of the last external fence in the level.

Let's define a relation $Employee < EmpId$ int , $Name$ char(20)>. We assume that $EmpId$ is a key column and the relation is sorted by this column. The relation contains the following records:

{(2,Bob),(3,Tom),(4,Mary),(6,Lucy),(7,Taylor),
(9,Andre),(10,Pat),(11,Pete),(12,Mary),(14,John),(16, Roger),
(20,Andy),(21,Jane),(23,Maria),(26,George),(27,Mike),
(28,Tim),(29,Pete)}

Figure 1 presents an example of the FA-index. The structure is maintained in four memory blocks: $B_1$, $B_2$, $B_3$ and $B_4$. Each block (except $B_1$) contains two memory pages. The tree consists of three levels. $L_1$ has one memory page which contains four entries. $L_2$ has two pages and $L_3$ consists of four pages. We assume that one page can contain at most four entries. For example, the page $P_{1,0}$ stores two external fences (2 and 14), one deleted item (4) and one record: (16,Roger).

### A. Database Operations

In this section, we describe typical operations which can be performed on the index: insertion, search, level merge, deletion and update.

*1) Insertion:* The insertion of a record into the relation $R$ works as follows. The record is inserted into the head tree. Then, if the head tree exceeds the predefined capacity, the level merge is invoked and the record is moved to the first level.

*2) Search:* Our approach supports effective querying with predicates defined on key values. Due to utilizing fences, we avoid fetching the entire blocks and go only through the pages. Let's assume that we want to find all employees for which $EmpId$ is between 3 and 10. We start in $L_1$ where we encounter the external fences: 2 and 14. Between them there is a deleted record 4. It means that 4 does not belong to our response even though it is included in the range. Then, we traverse to page $P_{2,0}$ which holds records 4 and 13. Both are not in the answer: 4 is deleted and 13 is out of range. The external fences in $P_{2,0}$ are in the required range. Thus, we move to pages: $P_{3,0}$ and $P_{3,1}$. From these pages we fetch the records which satisfy the search condition.

*3) Level merge:* The level merge is performed on two levels: $L_{i-1}$ and $L_i$. The process is similar to the merging of two sorted lists. The operation is invoked, when level $L_{i-1}$ is full, i. e. the number of items exceeds the fixed limit (see Algorithm 1).

During the merge execution, we ignore all internal and external fences in $L_{i-1}$ (line 4), because the tree will be reorganized and the fences will not be needed. In level $L_i$, we ignore only internal fences (line 5). The external fences are not modified, since they point to pages in level $L_{i+1}$ and this level will not be altered. As a result, two new levels are created: $L'_{i-1}$ and $L'_i$. In $L'_{i-1}$, we store only the external fences which point to the pages in $L'_i$. The second level contains all records from $L_{i-1}$ and $L_i$ (lines between 6 and 18). Apart from the normal entries, we write internal and external fences into $L'_i$. The external fences are copied from $L_i$ to $L'_i$. Additionally, for each page $P$ from $L'_i$, a new external fence is created. Its value reflects the key value of the first key of page $P$ and its $pid$ is set as $P$. The external fences are then stored in $L'_{i-1}$. The internal fence is created, when the first entry in the page is not the external fence (lines between 19 and 28).

As a result of the level merge process, we obtain two levels: $L'_{i-1}$ and $L'_i$ . When the new level $L'_i$ exceeds its capacity, it must be merged with $L_{i+1}$. This process will be repeated until any new level overruns its limit (lines from 30 to 31).

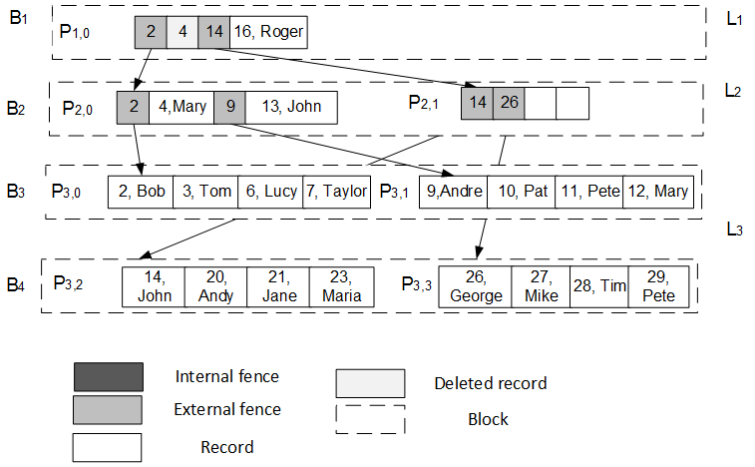


Fig. 1. FA-index

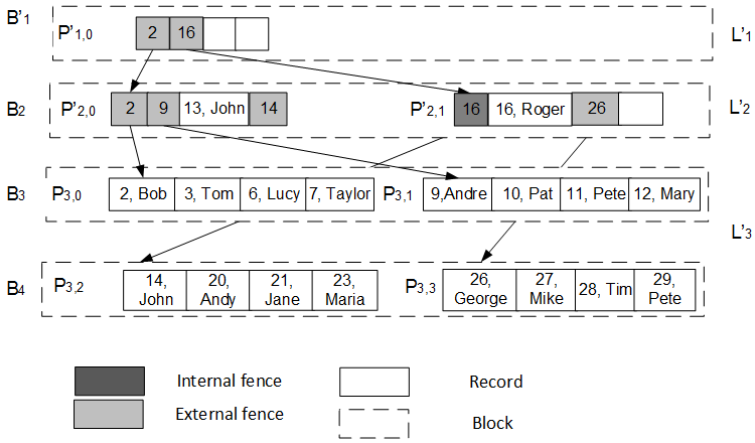


Fig. 2. Index structure after merging

In the example presented in Figure 1, the level $L_1$ stores already maximum number of entries. Please note that it maintains one record (16, Roger), one deleted record (depicted as 4), and two external fences (2 and 14). In level $L_2$, there are six entries, what means that it is possible to add two additional entries. Now suppose that the merge process of $L_1$ and $L_2$ is invoked. According to the description above, the process will be executed as follows.

1. All external fences in $L_2$ are rewritten to $L'_2$. It comes from the fact that the level $L_3$ will not be altered. So, the external fences in $L'_2$ point to the same pages as the ones in $L_2$.
2. The normal entries and the deleted entries from $L_1$ and $L_2$ must be inserted into $L'_2$ in an ascending order. In this step, we see an example of deletion. The deleted entry 4 will not be rewritten to $L'_2$, since there is already the record with the key 4 in $L'_2$. As a result, both entries are ignored in that level.
3. The internal fence 16 must be added at the beginning of page $P'_{2,1}$. It comes from the fact that the first entry of each page must be a fence.
4. After packing all required entries into $L'_2$, the external fences have to be created on $L'_1$ which point to the pages on $L'_2$.

The result of the level merge is illustrated in Figure 2. As it can be seen, we achieve some free space in $L'_1$. On the other hand, level $L'_2$ does not exceed the limited number of entries, so the merge process of $L'_2$ and $L'_3$ will not be started.

*4) Deletion:* The deletion is performed as follows. First, a record to delete must be found. For example, to delete an employee with $EmpId = 15$ (Figure 1), we look up the index. If we find the required entry, we insert a deleted record: $d'$ into the head tree. As in the case of insertion, the deletion is performed in the cascading way. So, when the level merge is executed, the deleted entry is rewritten to the lower level, until it does not reach the level where the corresponding normal entry is stored. Then, both entries are ignored.

*5) Update:* Update is implemented as a deletion of the old record followed by an insertion of the new one.

**Algorithm 1** Merge($Lvl_{i-1}, Lvl_i$)

```
Let e_{i-1} be the first entry in Lvl_{i-1}
Let e_i be the first entry in Lvl_i
while Lvl_{i-1} and Lvl_i are not empty do
    Skip all fences in Lvl_{i-1}
    Skip all internal fences in Lvl_i
    while e_{i-1}.type = DELETED and e_i.type = NORMAL and e_{i-1}.key = e_i.key do
        Skip both e_{i-1} and e_i entries
    end while
    if e_{i-1}.key < e_i.key then
        entryToInsert := e_{i-1}
        Let e_{i-1} be the next entry of Lvl_{i-1}
    else
        entryToInsert := e_i
        if e_i.type = External then
            lastFence := e_i
        end if
        Let e_i be the next entry of Lvl_i
    end if
    if The current Page P in Lvl'_i is empty then
        external.pid := P
        external.key := entryToInsert.key
        Write external to Lvl'_{i-1}
        if entryToInsert.type <> External then
            internal.pid := lastFence.pid
            internal.key := entryToInsert.key
            Write internal to Lvl'_i
        end if
    end if
    Write entryToInsert to Lvl'_i
    if Lvl'_i is full then
        Merge(Lvl'_i, Lvl_{i+1})
    end if
    Replace Lvl_{i-1} by Lvl'_{i-1}
    Replace Lvl_i by Lvl'_i
end while
```

## III. Cost Analysis

In this section, we consider the analysis of search and insertion cost for the FA-index and compare it with the B+tree index. The search cost is similar in both cases. It comes from the fact that both indexes are tree-like structures, so every level of the index must be accessed at most once while searching. The insertion cost of FA-index is caused by level merging. It is executed in bulks in contrary to the insertion into B+tree. In this analysis, we do not consider the erase cost, because it is carried out by the FTL which is independent of the proposed method. However, it is easy to suppose that the number of block erase would be bigger in the case of B+tree index, since this method engages more pages.

In the section, we have the following parameters:

1) $b$ - number of records in one page;
2) $r$ - width of the record in one page;
3) $s$ - page size;
4) $W$ - the write cost of one page;
5) $R$ - the read cost of one page;
6) $lvl$ - the number of levels in FA-index
7) $k$ - the logarithmic ratio between the number of entries of adjacent levels.

In table I, we compare the operation cost of FA-index and B+tree index.

### A. Search cost analysis

Let $|L_i|$ denote the number of entries in $L_i$. We assume the logarithmic ratio $k$ between the number of entries of adjacent levels. So, the following condition holds: $|L_{i+1}| = k * |L_i|$.

To find the required value, we need to traverse in the worst case through all levels. As every page is pointed by a fence, the search fetches only one page in each level. Thus, the time may be estimated as:

$$t_{search} = O(\log_k n) \tag{1}$$

### B. Insertion cost analysis

First, we consider the merge between two levels: $L_{i-1}$ and $L_i$. Let $m_i$ be a number of level merge operations after $n$ insertions. The maximum number of external and internal fences in level $L_{i-1}$ may be estimated as: $|L_i/b|$ and $|L_{i-1}/b|$, respectively. Thus:

$$m_i < \frac{n}{|L_{i-1}| - |L_i/b| - |L_{i-1}/b|} = \frac{b}{b-k-1} * \frac{n}{|L_{i-1}|} \tag{2}$$

By $N_i^{write}$ and $N_i^{read}$, we denote the number of entries to be written and to be read in all $m_i$ merges.

The level merge inserts all entries from $L_{i-1}$ and $L_i$ to $L'_i$. Apart from that, the fences are written to $L'_{i-1}$. Their number after $m_i$ merges is: $m_i * |L_{i-1}| - n$. Since the fences point to the pages in the level below, the number of entries in this level is: $b * (m_i * |L_{i-1}| - n)$. Thus, the number of entries to write is a sum of entries in $L'_{i+1}$ and $L'_i$ :

$$N_i^{write} = (1+b)(m_i * |L_{i-1}| - n) \tag{3}$$

During the merge process we read all entries from $L_i$ and $L_{i-1}$. We write them all to $L'_i$. Apart from that, we have to write the new external fences to $L_{i-1}$. Their number is $\frac{n}{b-1}$, because the external fence points to one page (every $b$ entries). Then, we have:

$$N_i^{read} = N_i^{write} - \frac{n}{b-1} \tag{4}$$

The upper bound of the merge cost per insertion may be estimated as:

$$t_{merge} = \frac{1}{n} * \sum_{i=1}^{lvl-1} \left(\frac{r * N_i^{write}}{W} + \frac{r * N_i^{read}}{R}\right) \tag{5}$$

$$< \frac{W+R}{W*R} * r * \sum_{i=1}^{lvl-1} \left(2 * \frac{1}{n} * (1+b)(m_i * |L_{i-1}| - n) - \frac{n}{b-1}\right)$$

TABLE I
THE I/O COST COMPARISON

| | Search Cost | Insertion Cost | |
|---|---|---|---|
| Index Name | Read | Read | Write |
| FA-index | $O(log_k n)$ | $O(\frac{k}{b-k} * log_k n)$ | $O(\frac{k}{b-k} * log_k n)$ |
| B+tree index | $O(log_b n)$ | $O(log_b n)$ | $O(log_b n)$ |

Let $\theta = \frac{W+R}{W*R}$, hence:

$$t_{merge} < 2 * \theta * r * (1+b) \sum_{i=1}^{lvl-1} ((\frac{m_i * |L_{i-1}|}{n} - 1) - \frac{1}{b^2-1})$$

$$< 2 * \theta * r * (1+b) \sum_{i=1}^{lvl-1} (\frac{m_i * |L_{i-1}|}{n} - 1) \quad (6)$$

Combining (6) with inequality (2) and $r * b < s$, we obtain:

$$t_{merge} < \theta * (2 * r + 2 * s) \sum_{i=1}^{lvl-1} (\frac{b}{b-k-1} - 1)$$

$$= 2 * \theta * (lvl-1) * (r+s)(\frac{b}{b-k-1} - 1) \quad (7)$$

$$< 2 * \theta * (lvl-1) * (\frac{s}{b} + s)(\frac{b}{b-k-1} - 1) \quad (8)$$

After some calculations, we obtain:

$$t_{merge} < 2 * \theta * s * (lvl-1) * (\frac{k+2}{b-k-1}) \quad (9)$$

Finally, from inequality (9), we can estimate the time of write and read as:

$$O(\frac{k}{b-k} * \log_k n)$$

## IV. EXPERIMENTS

In this section, we present several experiments which confirm the efficiency of our method. The experiments were conducted on the flash with the following characteristics. The NAND flash device is organised as a collection of blocks, with 64 pages per block. The page size is 2048 bytes. The page read time is 25$\mu s$, the page write requires 220$\mu s$ and the block erasing time is 500$\mu s$. For the experiments we use the table $Orders$ from TPC-H workload. We implement two clustered indexes. The first index is based on the traditional B+tree, the second one is the FA-index. We perform insertion into the B+tree in a traditional way or, if it is more profitable, in bulk loading. To optimize the storage we use the page-level FTL.

We employ three workload patterns:

1) Balanced Workload ($BWorkload$) - 50% read and 50% write operations.
2) Write Intense Workload ($WWorkload$) - 20% read and 80% write operations.
3) Read Intense Workload ($RWorkload$) - 80% read and 20% write operations.

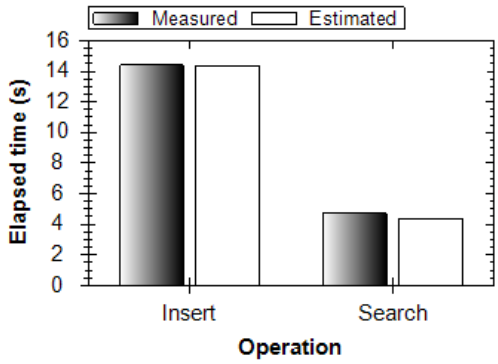


Fig. 3. Estimated vs. measured time (after 100000 operations)

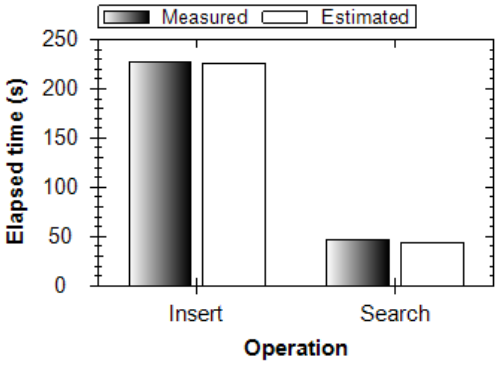


Fig. 4. Estimated vs. measured time (after 1mln operations)

The entire implementation is written in C language and compiled in Visual Studio 8.0. The experiments are conducted on Intel Core i5-2450M 2.5Ghz (4CPUs) with 4GB RAM using Windows 7 Professional.

In the first experiment, we inserted 100000 records. After that we performed 1mln point search operations using FA-index. By the point search we mean an operation which seeks a single value in the index. Normally, the point search requires traversing from the root to the leaf of the index. In the Figures 3 and 4, we compare the measured time with the expected time obtained using the algorithms simulating the FA-index. In the second experiment, we compare the clustered index based on the traditional B+tree with the FA-index. Figures 5 and 6 present the elapsed time after 100000 and 1 mln operations, respectively. The operations are distributed according to the workload patterns described above.

From the experiments, we can conclude that the FA-index outperforms the traditional methods based on B+tree for all

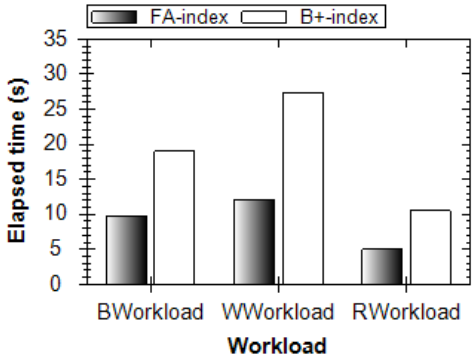


Fig. 5. Comparison FA-index with B+-tree (after 100000 operations)

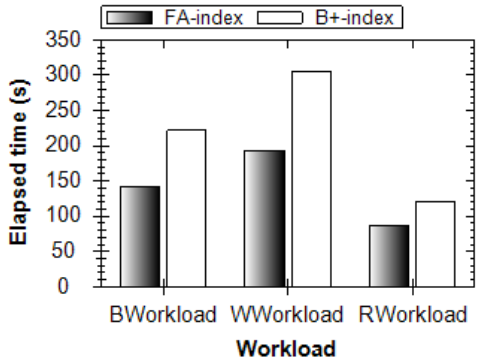


Fig. 6. Comparison FA-index with B+-tree (after 1mln operations)

workload patterns. It is due to the fact that the parameter $k$ may be changed depending on the workload. For the write intense workload, it is better to create the FA-index with more levels in order to avoid level merging. On the other hand, for the read intense workload, the number of levels may be smaller, what decreases the number of the traversing pages.

## V. Conclusions

The popularity of flash disks has grown dramatically in the last years. They are characterized by the asymmetry in speed of read and write operations, what plays an important role in the design of database storage methods.

In this work, we present the new approach to the storage of the clustered index for mobile databases. The proposed FA-index is write and read efficient in the flash memory context. The read efficiency is achieved due to the logarithmic searching time. The write efficiency comes from the fact that we do not reorganise the whole structure during inserting. Apart from that, our block oriented approach causes the good exploitation of the flash memory and decreases the number of expensive erase operations. The theoretical and experimental analysis show, that the FA-index outperforms the traditional clustered B+tree index.

## Acknowledgment

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

## References

[1] Cho, H., Shin, D., Eom, Y.I.: Kast: K-associative sector translation for nand flash memory in real-time systems. In: DATE'09. (2009) 507–512
[2] Park, C., Cheon, W., Kang, J., Roh, K., Cho, W., Kim, J.S.: A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. ACM Trans. Embed. Comput. Syst. **7**(4) (2008) 1–23
[3] Kang, J., Jo, H., 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
[4] Lee, S.W., Park, D.J., Chung, T.S., Lee, D.H., Park, S., Song, H.J.: A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans. Embed. Comput. Syst. **6**(3) (2007) 18
[5] Oh, G., Kim, S., Lee, S.W., Moon, B.: Sqlite optimization with phase change memory for mobile applications. Proc. VLDB Endow. **8**(12) (2015) 1454–1465
[6] Kang, W.H., Lee, S.W., Moon, B., Oh, G.H., Min, C.: X-ftl: Transactional ftl for sqlite databases. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. SIGMOD '13, New York, NY, USA, ACM (2013) 97–108
[7] 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)
[8] Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. Proc. VLDB Endow. **3**(1-2) (2010) 1195–1206
[9] Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., Singh, S.: Lazy-adaptive tree: An optimized index structure for flash devices. Proc. VLDB Endow. **2**(1) (2009) 361–372
[10] Athanassoulis, M., Ailamaki, A.: Bf-tree: Approximate tree indexing. Proc. VLDB Endow. **7**(14) (2014) 1881–1892
[11] Kwiatkowski, K., Macyna, W.: CFTL - flash translation layer for column oriented databases. In: Intelligent Information and Database Systems - 5th Asian Conference, ACIIDS 2013, Kuala Lumpur, Malaysia, March 18-20, 2013, Proceedings, Part I. Lecture Notes in Computer Science 7802, Springer (2013) 146–155