World Wide Web (2017) 20:1017–1045
DOI 10.1007/s11280-016-0426-9

Tide-tree: A self-tuning indexing scheme for hybrid storage system

Sheng Wang[1] · Xiaolin Qin[2] · Zhifeng Bao[1] · Bohan Li[2]

Received: 17 September 2016 / Revised: 5 December 2016 /
Accepted: 9 December 2016 / Published online: 21 December 2016

Abstract Main memory index is built with the assumption that the RAM is sufficiently large to hold data. Due to the volatility and high unit price of main memory, indices under secondary memory such as SSD and HDD are widely used. However, the I/O operation with main memory is still the bottleneck for query efficiency. In this paper, we propose a self-tuning indexing scheme called Tide-tree for RAM/Disk-based hybrid storage system. Tide-tree aims to overcome the obstacles main memory and disk-based indices face, and performs like the tide to achieve a double-win in space and performance, which is self-adaptive with respect to the running environment. Particularly, Tide-tree delaminates the tree structure adaptively with high efficiency based on storage sense, and applies an effective self-tuning algorithm to dynamically load various nodes into main memory. We employ memory mapping technology to solve the persistent problem of main memory index, and improves the efficiency of data synchronism and pointer translation. To further enhance the independence of Tide-tree, we employ the *index head* and the level address table to manage the whole index. With the *index head*, three efficient operations are proposed, namely index rebuild, index load and range search. We have conducted extensive experiments to compare the Tide-tree with several state-of-the-art indices, and the results have validated the high efficiency, reusability and stability of Tide-tree.

✉ Zhifeng Bao
zhifeng.bao@rmit.edu.au

Xiaolin Qin
qinxcs@nuaa.edu.cn

Sheng Wang
sheng.wang@rmit.edu.au

Bohan Li
bhli@nuaa.edu.cn

[1] RMIT University, Melbourne, Australia

[2] School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

Keywords Self-tuning index · Memory map · Pointer swizzling · Hybrid storage

1 Introduction

Main memory index has been a mainstream direction of database for many years, and the achievements are plentiful. Regarding eliminating I/O operations between main and secondary memory (such as HDD (Hard Disk Drive) and SSD (Solid-State Disk), etc.), the main memory index is very efficient, but it is not reusable because of its volatility. Volatility is critical, because one of the desired features of a DBMS is the ability to retain its data even in the presence of errors such as power failures [11]. Another challenge of main memory index is that the main memory's space is always restricted, the performance will degrade greatly if the main memory is not adequate to accommodate the whole index. Therefore, lots of environmental factors impel the main memory index to be self-tuning to meet many challenges [4].

Table 1 shows a comparison of search performance, insertion performance and unit price among RAM, SSD and HDD. Note that we take the most common index B^+-tree to get these results. We observe that B^+-tree gets the best performance in main memory, worse in SSD and the worst in HDD. Main memory is still the best running environment for index in terms of query efficiency. However, it will cost much computation and storage resource to create the main memory index due to the volatility and restricted space. Furthermore, the unit price of main memory is much higher than SSD and HDD, so most commercial products and research work employ disk-based (including hard disk and flash disk) indexing schemes to achieve a more stable performance [10, 12, 30, 33].

In order to bridge the gap between CPU processing and I/O operation, hierarchical memory structure is designed and widely employed in modern computer systems, such as embedded sensors, mobile platforms, personal computers and enterprise servers. As the capacity of main memory increases, more and more platforms choose to utilize it to improve the performance, e.g. Spark [39] improved Hadoop [32] by Resilient Distributed Datasets [38] in main memory to process data instead of disk and got high performance.

Obviously, a fixed indexing scheme may have various performance under different computation frameworks. Both main memory and disk-based indexing schemes cannot meet the demand for diversified environments. Therefore, designing an index which can meet most scenarios is a challenging task. Once the environment changes, such as running devices, we should turn the index to achieve the optimal performance. So lots of self-tuning indices for hybrid storage system were proposed. However, to the best of our knowledge, most of them ignore the utilization of main memory. Furthermore, it remains an open problem to have self-tuning indices for RAM/Disk-based hybrid storage system, due to the persistence

Table 1 Performance and price comparison among three devices

Device	Search(s)	Insert(s)	Price($/GB)
RAM(Kingston)	0.223	0.478	9.75
SSD(OCZ SATA)	0.576	1.478	0.558
HDD(WDC)	1.324	2.434	0.081

problem of main memory indexing schemes. That is why researchers prefer the secondary memory to ensure integrity of index. All existing indexing schemes, including those in commercial products and research prototypes, suffer from the drawback that they are not self-tuning for RAM/Disk-based hybrid storage systems. For instance, in main memory database, e.g. Oracle's TimesTen [20] and Microsoft's Hekaton [8], database administrators are forced to make explicit tuning decisions and requests [14].

In order to achieve a double-win between efficiency and space, we propose a self-tuning indexing scheme for hybrid storage systems which utilize both main and secondary memory adaptively. The main contributions of this paper can be summarized as follows:

- We propose an efficient self-tuning algorithm which delaminates the tree adaptively based on storage devices. Tide-tree improves the performance of disk-based index and overcomes the shortcomings of main memory index to some extend.
- Based on the memory mapping technology, we propose a persistence method to maintain efficient synchronization of index between main and secondary memory. We also apply an improved pointer swizzling technology, which is able to access data quickly, as well as abandon the traditional translation table.
- We introduce two efficient operations and improve the range search operation, then implement the prototype in Linux. We empirically evaluate Tide-tree in comparison with FD-tree and other state-of-the-art schemes.

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 presents the design of Tide-tree, including the structure, storage method and self-tuning strategy. Section 4 gives the key operations and performance analysis of Tide-tree. Section 5 reports the experimental evaluation of the performance of Tide-tree. Finally, we conclude in Section 6.

2 Related work and preliminary

This section reviews the related work on index under different storage devices, including main memory and secondary memory. Then we introduces the self-tuning indices for various devices. Table 2 lists some existing indexing structures and compare the commonality and difference with Tide-tree.

Table 2 Commonality and difference between Tide-tree and existing indexing structures

	Main Memory	HDD	SSD	Self-tuning	Reusable
Tide-tree	✓	✓	✓	✓	✓
FD-tree [25]			✓	✓	✓
B^+-tree(ST)[29]			✓		✓
HybridB-tree [17]		✓	✓	✓	✓
DPT [19]	✓		✓		✓
CSB^+-tree [31]	✓				
ART-tree [23]	✓			✓	
FB-tree [18]			✓		✓
BF-tree [2]	✓	✓			

2.1 Indices on different storage devices

According to the type of running storage devices, we can divide existing indices into main memory index and disk-based index. The former can achieve better performance for running in RAM directly without I/O operation. T-tree [22] is a balanced index tree data structure optimized for cases where both the index and the actual data are fully kept in memory. As a variant of B+-Tree [5], CSB$^+$-tree [31] (cache sensitive B$^+$-tree) stores all the child nodes of any given node contiguously and keeps the address of the first child in each node only. CSB$^+$-tree overcomes the performance degradation problems of B$^+$-tree running in main memory. Viktor et al. [23] proposed an adaptive main memory index based on Radix-tree called ART-tree which sets different size of node dynamically and owns small footprint and high performance. It chooses compact and efficient index structure to reduce the footprint, which is comparable with hash index in terms of efficiency. However, the restricted space under small RAM and big data processing scenario is the main challenge of main memory index. To avoid the restricted space problem when the database cannot fit into main memory, Goetz et al. [14] adopt improved buffer pool and swizzling pointers to alleviate the problem of main memory database performance degradation effectively.

Disk-based index is widely used in commercial databases and search engines for high reusability [9, 36], especially with the emergence of SSD. To further improve the processing efficiency and reducing the gap with main memory index, FB-tree [18] proposed by Martin et al. is optimized for modern SSDs. It is the combination of an adaptive B$^+$-tree, a storage manager and a buffer manager. Lazy-Adaptive Tree [1] is a novel index structure designed to minimize the accesses to flash. It amortizes the cost of node reads and writes by performing update operations in a lazy manner using cascaded buffers, thereby minimizing response time. To achieve compromise between performance and space, BF-tree [2] uses Bloom filter [28] to compact the leaf nodes, it can be built and maintained entirely in main or secondary memory. However, BF-tree only supports such two patterns and is still greatly restricted by main memory.

Thomas et al. [19] extended the work of Boehm et al. [3] and achieved a durable main memory prefix-tree called DPT (Durable Prefix Tree) in the SIGMOD 2011 Programing Contest.[1] This is a rare scheme that utilizes main and secondary memory both. DPT employs a coalesced cyclic log mechanism which collects single log records in a write buffer before flushing to disk to achieve maximum throughput. When the system crashes, DPT can use the operation sequence in disk to rebuild the main memory index quickly. However, this scheme applies group committing method which just includes the insertion and deletion. It needs to review the index file twice to read all these operations, which leads to low efficiency. In other words, the DPT just implements the persistence of updating operation but not the index structure.

Buffer technology is another common tool employed to optimize the performance of index by loading those nodes which are accessed a lot. For SSD, Yang et al. [37] focused on buffer schemes for tree indexes on SSDs especially according to the features that the root and internal nodes have higher read frequencies than leaf nodes have. Lee et al. [21] improve the building efficiency by buffering the updating operation in SSD, it can significantly reduces the number of write operations to a flash memory when constructing a B-tree. However, the main difference compared with their work is that those work use SSD as main storage device and do not use main memory at all, while we take main memory which is

[1] http://db.csail.mit.edu/sigmod11contest/

faster than SSD as storage device. Meanwhile, we can maintain the index structure in memory efficiently and can also use less time to load index from external disk. Moreover, Long et al. [27] propose and evaluate a three-level caching scheme that adds an intermediate level of caching for additional performance gains.

2.2 Self-tuning indices

Existing self-tuning indices [4, 13] can be categorized into two types. One is to design physical structure which can achieve self-tuning according to different size of devices and workloads. Combining the B^+-tree with the B^+-tree(log) [35], the B^+-tree(ST) [29] proposed by Suman et al. is a self-tuning B^+-tree index which can reduce the updating overhead of B^+-tree under different storage devices and workloads while keeping high search performance. HybridB tree [17] proposed by Jin et al. mainly changes the leaf nodes of B^+-tree to adapt to the SSD/HDD hybrid storage system, it distributes the leaf pages in a huge leaf node among HDD and SSD to improve searching efficiency and reduce updating overhead. However, these schemes only utilize secondary memory and are more suitable for sensors and wireless mobile. For those personal computers and enterprise servers with relatively large RAM, the utilization of main memory is too low, so the performance is low. FD-tree [24, 25] proposed by Li et al. is a multi-layered tree index for Flash SSD. It maintains a small B^+-tree as the head of whole index, and organizes other pages below the head as arrays. It minimizes the number of small random writes and limits these random writes within a small area, while maintaining a high search efficiency. Although we can turn the size of buffer pool and *Head tree*, it cannot be a purely main memory index when the RAM is sufficient. Additionally, the size of *Head tree* is fixed until recreating the index, so FD-tree choose small block of main memory to store the *Head tree* to guarantee fitting in most environment. The main difference with FD-tree is that the index is stored in SSD while our scheme stores index in main memory to improve the query efficiency, when the space of main memory is not enough to store whole index, we will adaptively load part of index structure into external index.

Another kind of self-tuning indexing scheme chooses one from many indices such as the B^+-tree, Hash-index to fit the various environment. Database cracking [16] has been proposed as a solution that allows on-the-fly physical data reorganization, as a collateral effect of query processing, aims to continuously and automatically adapt indices to the workload at hand [15]. Goetz et al. [13] combined the efficiency of traditional B^+-tree creation with the adaptive and incremental behavior of database cracking, instead of traditional offline analyzing and detecting index turning scheme. Felix et al. [15] proposed a stochastic cracking technology based on database cracking. Since it uses each query as a hint on how to reorganize data, but not blindly, this approach is able to gain resilience and avoid performance bottlenecks by deliberately applying certain arbitrary choices in its decision-making. These schemes do not take the storage device into account at all, because they focus on the self-tuning strategies.

3 General design of tide-tree

Based on FD-tree and memory mapping technology, we propose a self-tuning indexing scheme called Tide-tree. This section will discuss the shortage of FD-tree and our motivation first, then present the structure design of Tide-tree, persistence of index structure and self-tuning strategy, respectively.

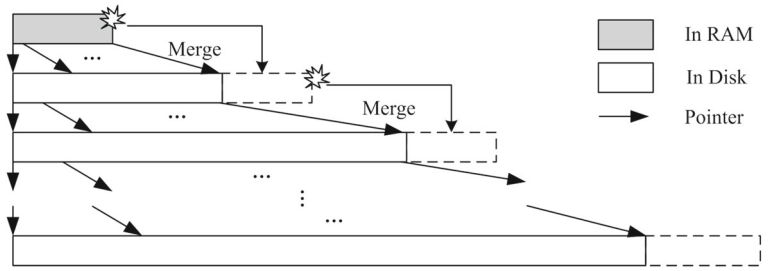

Figure 1 Architecture of FD-tree

3.1 Motivation and problems

Figure 1 shows the architecture of FD-tree. The star of each level means that this level reaches the capacity and merges to next level so to transform random writes into sequential ones, which can reduce the updating overhead. The reason why we choose FD-tree to improve is based on two major stands. On one hand, as a disk-based index, FD-tree guarantees high query efficiency compared with B^+-tree under SSD or HDD, the hierarchical tree structure makes each level to be independent to manage. On the other hand, since the footprint is quite small, it can accommodate more keys into main memory. B^+-tree [5] is widely used in lots of database for its good performance, but it is updating overhead is quite expensive. Additionally, the role of one node in the tree always changes, it will cost much to decide the node whether in main memory or not. Therefore it is hard to maintain stable performance for RAM/Disk-based hybrid storage system.

From Figure 2 above, we can see that for two kinds of disks, the bigger the accessed block is, the higher the bandwidth is. FD-tree chooses to write to disk at a size of page such

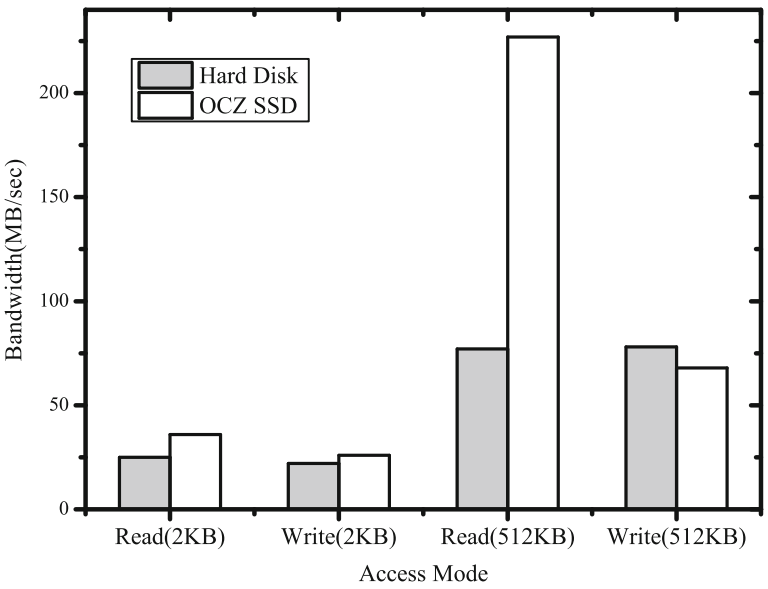

Figure 2 Bandwidth under different type of access

Table 3 Memory footprint of different indices

Indices	Memory footprints			Height
	Single Node	All Internal nodes	All Leaf nodes	
FD-tree	2k	1.2M	80M	3
B^+-tree	64	28.4M	130.6M	7
CSB^+-tree	64	12.8M	130.6M	6

as 2KB, if we just increase the size of block write to secondary storage every time, it will improve the updating efficiency further.

Table 3 provides a view of the footprint of different indices. It shows the occupied space of single node, internal nodes and leaf nodes, height of tree, respectively. The number of index entries is set to be 10^7. We can find that the leaf nodes occupy the most of the whole index. If we just load the internal nodes of the tree into main memory, it would not occupy too much space but can improve the search efficiency significantly compared to disk-based index. Specially, loading much less than half of the tree into main memory will improve the efficiency considerably larger than half.

The main objective of our scheme is building a self-tuning index which utilizes the main and secondary memory both to realize double-win situation of efficiency and space. However, the biggest challenge lies in the storage and self-tuning strategy, as described below:

- Storage and loading of index structure, that is to say how to make the whole index durable in secondary memory;
- Efficient self-tuning algorithm to ensure a double-win situation between efficiency and space.

3.2 Architecture of tide-tree

As Figure 3 shows, the proposed scheme is a hybrid tree index with l levels, denoted as $L_0 \tilde{} L_{l-1}$, respectively. It consists of three parts, the first part is *Head tree*, denoted as L_0. We take CSB^+-tree to replace the B^+-tree of FD-tree for its higher query efficiency and smaller footprint, which is more appropriate for main memory. But the node size of *Head tree* is not limited by the cache line. The middle part $L_1 \tilde{} L_{m-1}$, $(m < l)$ are those levels which stay in main memory when there exists sufficient spaces, the $L_m \tilde{} L_{l-1}$, $(m < l)$ below are the levels which stay in secondary memory. L_i, $0 < i < l$ are all sorted runs. We call L_i, $0 \leq i < l-1$ as internal level, and call L_{l-1} as leaf level. When the whole tree fits into main memory,

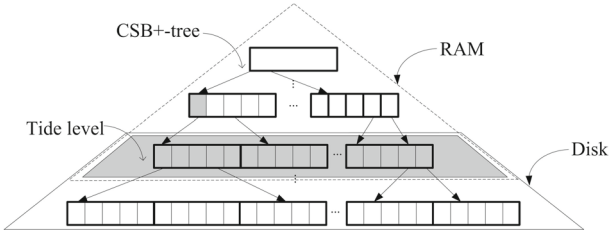

Figure 3 Architecture of Tide-tree

$m = l$; otherwise we have $m < l$, i.e., these levels locate in main and secondary memory partly. When $m = 0$, the whole tree locates in secondary memory, this is the worst case in terms of efficiency. However, when main memory can partially accommodate a level, we choose to load this level locally to make full use of main memory. We set the ratio of a loaded level to be $u_{RAM} \in [0, 1]$.

The most essential quality of Tide-tree is that it can switch between main memory index and RAM/Disk-based index adaptively, just like the tidal lane which could be turned according to the two-way traffic. That is why we name the proposed scheme as Tide-tree. Since the query operations to the tree such as search and insertion have to traverse from root to leaf through every level, hierarchical structure can guarantee stable and efficient query for unpredictable random query. Next we will talk about the details of structure of node, mode turning of node and *index head*.

3.2.1 Structure of node

The Nodes of Tide-tree are composed of many entries. Each entry includes index keys, pointers or row identifiers, which can be specified by 2 bits. Each entry takes up two words. Figure 4 above illustrates the details of node and entry.

The entries are divided into four types, including $Entry_{normal}$, $Entry_{filter}$, $Fence_{external}$, $Fence_{Internal}$. They are all composed of key, type, offset. The details are showing as follows:

$Entry_{normal}$: an index entry containing the identifier;
$Entry_{filter}$: an index entry waiting to be deleted;

$Fence_{external}$: an external pointer directing to the node in next level;

$Fence_{Internal}$: an internal pointer directing to the node in next level when the first entry is not an external fence;

The *offset* represents the row identifier pointing to data table in $Entry_{normal}$ and $Entry_{filter}$. It represents the pointer of child node in $Fence_{external}$ and $Fence_{Internal}$, we use the *type* to specify. We know that the key is inserted to leaf level eventually by many times of merge operation.

To tell where the node locates easily, we give a label to each node, called as *Dirty State* which indicates that all internal levels or any level L_i $(0 \leq i < l - 1)$ not only stores the fences but also the index entries when it is true, otherwise just stores the fences.

This label will help show the existence of index entries in internal levels. When the Tide-tree is not in dirty state, the point search is stable and the range search is most efficient,

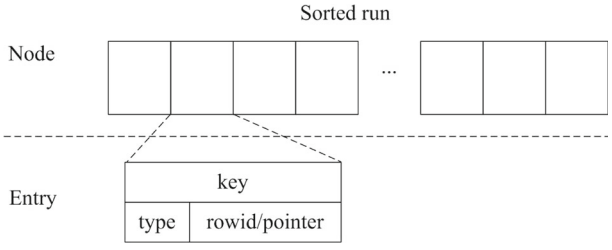

Figure 4 Structure of node and entry

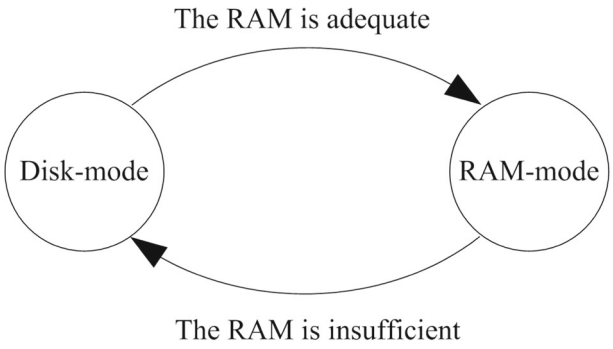

Figure 5 Mode switching of node

otherwise the query will be much complicated. However, with labeling the dirty state in *index head*, we can simplify the range search and rebuild operations, as illustrated later in Section 4.

3.2.2 Two modes of nodes

Every node except in the *Head tree* can switch between RAM and Disk, it depends on whether the RAM is sufficient to accommodate it. We introduce two modes here, *RAM-mode* and *Disk-mode* respectively. As Figure 3 shows in last section, nodes in the gray area of index tree are staying in *RAM-mode*, while nodes in remaining area below are staying in *Disk-mode*. Figure 5 illustrates the details of mode switching. We do not mark every node's mode for wasting space; instead, we just use u_{RAM} to mark the whole level. That is because all nodes in a level are sorted, if one node is loaded in main memory, the former nodes must be staying in *RAM-mode*. When $u_{RAM} = 0$, all nodes in this level are staying in *Disk-mode*. $u_{RAM} = 1$ means staying in *RAM-mode*. $0 < u_{RAM} < 1$ means that previous nodes in this level are staying in *RAM-mode*, and the number is $u_{RAM} \cdot |L_i|$. The mode of each level is set during index creation, load or rebuild, mainly caused by the change of environment.

3.2.3 Index head and level address table

Last section uses u_{RAM} to indicate the mode of each level, in order to organize the index structure well, there also exists other important information to describe the index. Hence we propose a structure called *index head* stored in the head of index file to record important parameters. The *index head* includes the global parameters and level address table(LAT), it will stay in main memory all the time, which is similar to the data dictionary in database.

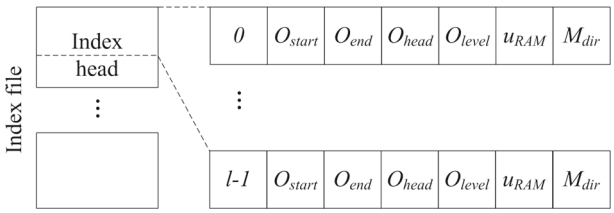

Figure 6 *Index head* and LAT

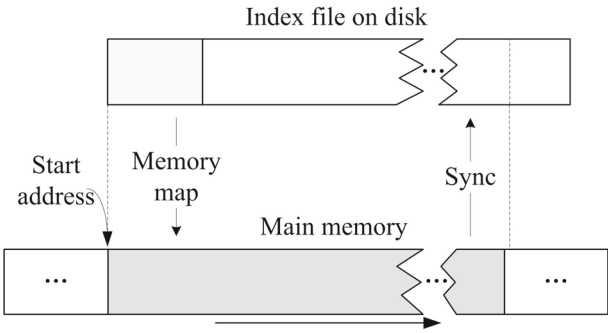

Figure 7 Persistence of Tide-tree

The global information of index includes the details of index tree and index file such as dirty state of the whole tree, the root node, the offset, etc. The LAT stores local information of each level, it is composed of l arrays (O_{start}, O_{end}, O_{head}, O_{level}, u_{RAM}, M_{dirt}), which represent the starting offset, ending offset relative to the index file, the offset of head node, increasing offset relative to the level and the loading ratio of level, respectively. M_{dir} indicates whether existing index entries in the level, it is helpful in the range search and rebuild operation next. Since we need to access *index head* frequently, we set the size of *index head* as multiple of cache line size and align the whole block to guarantee retaining in cache. Figure 6 illustrates the *index head* and LAT.

3.3 Persistence of tide-tree

In order to assure reusability, we choose to store the whole Tide-tree into secondary memory. As shown in Figure 7, we apply memory mapping technology[2] to map the index file into main memory partly or entirely. Then we can access the index file just like in main memory by pointers without calling *read() or write()*.

3.3.1 Memory mapping and management

Index file in disk maintains consistency with main memory by memory mapping technology, which is originally proposed in [34]. Once the mapping relation is built, the index structure will establish a copy to the index file. Changes to the index in main memory will eventually flush to disk efficiently with calling function $msync()$.

Dealing with large file processing problem, the memory mapping technology supported by operating system(Linux and Windows both support) is a very effective method without utilizing buffer pool. The memory mapping is far more efficient than traditional file operations which read or write a block or page, while memory mapping technology map whole file to memory from disk directly and avoid accessing by small blocks. All the changes towards the mapping area will be flushed to external device and saved efficient. Hence memory mapping technology is highly efficient especially for large file, like our index file.

[2] http://linux.die.net/man/2/mmap

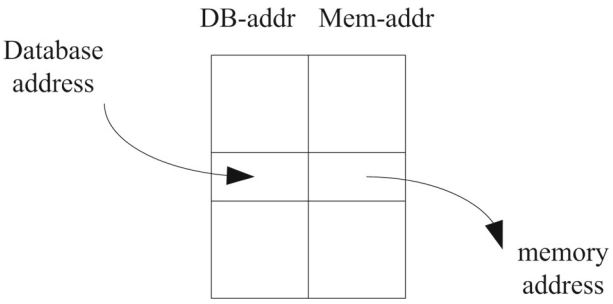

Figure 8 The translation table

As we know, some main memory databases such as Tokyo Cabinet[3] and Redis[4] also employ this technology to store the data table. This technology is also widely used in mass data processing, data sharing and graph computation [26], etc. However, it has not been used in storage of complicated index structure.

Since we cannot call system tools to manage the mapping area, so we have to define a memory mechanism all by ourselves. We take an incremental allocating and chained recycling method to guarantee the compact of mapping area so to accommodate nodes as much as possible. Firstly, we use LAT to record the current offset of each level, initialize $LAT\,[x]\,.O_{offset}$ $(0 \leq x < l)$ as 0. $LAT\,[x]\,.O_{offset} += len$ if the size of block allocated is len. Another key point of memory mapping technology is that we must align the index file to main memory block, otherwise the mapping length will be restricted.

3.3.2 Improved pointer swizzling technology

The access to index includes read and write. Compared with traditional method, the node may stay in main memory or not, which means we need a method to ensure the completeness of accessed data and would not occur segment fault.

As shown in Figure 7, the mapping can be divided into two types according to the start address. One is that the start address is fixed and must be the same in every mapping, the universality is very poor. Another one is allocating randomly according to the operating system, the start address will be various in every mapping. In consideration of transferability and commonality, we choose the latter. The index file will store lots of pointers, and direct to the nodes located in mapping area or not if we just store the pointers directly. When taking the random way to map, the start address is changed after the reboot of mapping, the origin pointers may direct to the area out of mapping, therefore it will cause segment fault undoubtedly.

To transform the pointers between main and secondary memory, traditional databases always apply the translation table to turn the database address into equivalents in memory, shown as Figure 8. To avoid translation repeatedly, the pointer swizzling technology [7] was proposed. But the swizzling technology still needs a translation table which wastes much time and space.

[3] http://fallabs.com/tokyocabinet/

[4] http://redis.io/

To overcome this problem, we propose an improved pointer swizzling technology which do not need a translation table anymore. More specifically, we employ to store the offset of pointers relative to the start address. When the index file is mapped into main memory, we record the start address $base$ into *index head* first, then we need to subtract $base$ from the pointer and store it. For instance, the pointer is ptr, then the address need to store is $(ptr - base)$. When we need to access the data which the pointer directs to, we just add the $base$ of this mapping to *offset* to ensure that all the accessing data locates in mapping area. Although lots of additions and subtractions may occur during index creation and query, but through the experiment we find that compared with running in main memory directly, the proposed scheme is almost the same in terms of query efficiency. Because the addition and subtraction are very efficient and will not affect the index's performance. The benefit of improved pointer swizzling technology is that we will not need to maintain a large address table which store all the address of nodes and will cost much space in main memory, it also need to load the data from main memory to cache which also cost much time, while we only need to do additions and subtractions whose cost can be ignorable.

When the pointing block is located in main memory, we can access it just by the offset, otherwise we can call the traditional way to read a certain length of block from the secondary memory by offset too. Algorithm 1 shows the details of reading a block according to the offset and size.

Algorithm 1 *read_block(offset, size)*

Require: *offset* : the offset of block in the index file, *size* : the size of block;
Ensure: *node* : the block pointer in main memory;

```
1: if offset >= IndexHead.MapLen then
2:     node = malloc(size);
3:     fseek(IndexHead.fp, offset);
4:     fread(IndexHead.fp, node, size);
5: else
6:     node = offset + IndexHead.base;
7: end ifreturn node;
```

3.4 Self-tuning Strategy of Tide-tree

This section discusses two optimal models first, namely, the optimal node size and optimal level ratio. Then, we introduce the mode setting algorithm which sets different mode adaptively for index nodes to achieve self-tunning of Tide-tree.

We denote the number of index entries as N, the number of entries in each level as $|L_i|\,(0 \le i < l)$, and k_i is the logarithmic size ratio between adjacent levels L_i, L_{i+1}, so $k_i = \frac{|L_{i+1}|}{|L_i|}, 0 \le i < l-1$. The performance is optimal when $k_i\ (0 \le i < l-1)$ are equal to each other [25], so we use k to represent the unified level ratio. The page size is denoted as B, the number of entries in a node is f, each entry occupies $|e|$, so $f = \frac{B}{|e|}$. Refer to the paper [31], the number of leaf entries stored in L_0 is:

$$|L_0| = |H| \cdot \frac{u\,(f-2)\,(u\,(f-2t+1)-1)}{B} \tag{1}$$

Where $|H|$ is the size of *Head tree*, in order to ensure the alignment, we set $|H| = 1024$ bytes, t is the segment number, u is the utilization of node. We can notice that when

$N \leq |L_0|$, the number of entries is so small that the index tree is totally a CSB$^+$-tree. We start employing under layers when $N > |L_0|$. To ensure that the number of fences fetched from will not exceed the threshold of L_i, so:

$$|L_{i+1}|/f + |L_i|/f < |L_i|, 0 \leq i < l-1 \tag{2}$$

The $|L_{i+1}|/f$, $|L_i|/f$ are the max number of external and internal fences of L_i, respectively. So the level ratio should meet following criteria:

$$k < f\text{-}1 \tag{3}$$

During the index creation, we need to create the index file with initial size first. Therefore, we should calculate the size of the index occupied. Since the leaf level stores all the keys and row identifiers, the footprint of leaf level is:

$$|L_{l-1}| \cdot |e| \geq \frac{N \cdot B}{f} \tag{4}$$

The height l can be calculated as:

$$l = \left\lceil \log_k \frac{N}{|L_0|} \right\rceil + 1 \tag{5}$$

And the whole footprint of Tide-tree is:

$$|L| = |e| \cdot |L_{l-1}| \left(\sum_{i=0}^{l-2} \frac{1}{k^i} \right) + |H| + |IH| \tag{6}$$

Where $|IH|$ represents the size of *index head*. The size of index file is initialized according to (6). When the index file is not enough to accommodate those keys of insertion afterwards, we need to close the mapping relation first, extend the file and map it to main memory again.

3.4.1 Optimal node size and level ratio

For the accessed difference between main and secondary memory, the performance may vary from different sizes of node. However, Tide-tree needs to turn frequently according to the environment. To guarantee the optimal query efficiency and stability of the index without increasing the overhead caused by index turning, for L_i $(0 < i < l)$, we take nodes with unified size.

Big node can reduce the height and the number of nodes traveling from root to leaf, but also increase the overhead of searching inside the node. Additionally, the bigger the node is, the larger block reads from the index file. These two competing factors lead to the following analysis. The query starts from the *Head tree*, we get the fence pointing to next level if the key does not exist, and search inside the node for every level until finding the key. The search overhead can be denoted as follows:

$$\begin{aligned} cost_{search} = & (m + u_{RAM} + l_{head}) \cdot R_R(B) \\ & + (l - m - u_{RAM} - 1) \cdot R_D(B) \\ & + (l + l_{head} - 1) \cdot C_{node}(B) \end{aligned} \tag{7}$$

Where $R_R(B)$ and $R_D(B)$ are the overhead of accessing the node under main and secondary memory, respectively. $C_{node}(B)$ is the overhead searching inside the node in main memory. The height of *Head tree* l_{head} is $\lceil \log_f |L_0| \rceil$.

Level ratio k also decides the performance of Tide-tree. The bigger the k is, the more normal entries will insert to next level, so the merging overhead of two levels will be high. But the smaller the k is, the more often the merging is, the bigger the height of tree is and the search overhead is also very high. Obviously, the optimal k is another important factor we need to take into account. Like the node size, we will discuss a framework to evaluate the level ratio too. This framework just improves the model of FD-tree and uses statistics to choose optimal k. The difference is that our overhead is more diverse, not only including the overhead of main memory but also the secondary memory and mapping. The evaluation of insertion overhead is described as Algorithm 2.

Algorithm 2 Level Ratio Estimation

Require: $|L_i|$, $|L_i^{'}|$: the capacity and the current cardinality of level i, respectively;
Ensure: $cost_{insert}$: the overhead of insertion;

1: $i \leftarrow 0$,$numInsert \leftarrow 0$;
2: $numRead_R \leftarrow numWrite_R \leftarrow numRead_D \leftarrow numWrite_D \leftarrow 0$;
3: **while** $i < l - 1$ **do**
4: $\quad numInsert \leftarrow numInsert + (|L_0| - |L_0^{'}|)$;
5: $\quad |L_0^{'}| \leftarrow |L_0|$;
6: $\quad i \leftarrow 0$;
7: $\quad$ **while** $|L_i^{'}| \geq |L_i|$ **do**
8: $\quad\quad numberRAM \leftarrow |L_i^{'}| \cdot L_i.u_{RAM} + |L_{i+1}^{'}| \cdot L_{i+1}.u_{RAM}$;
9: $\quad\quad numRead_R \leftarrow numRead_R + numberRAM$;
10: $\quad\quad numRead_D \leftarrow numRead_D + |L_i^{'}| + |L_{i+1}^{'}| - numberRAM$; //the number of entries lie in disk;
11: $\quad\quad |L_{i+1}^{'}| \leftarrow |L_{i+1}^{'}| + (|L_i| - |L_i^{'}|)$;
12: $\quad\quad |L_i^{'}| \leftarrow |L_{i+1}^{'}|/f$;
13: $\quad\quad numberRAM \leftarrow |L_i^{'}| \cdot L_i.u_{RAM} + |L_{i+1}^{'}| \cdot L_{i+1}.u_{RAM}$;
14: $\quad\quad numWrite_R \leftarrow numWrite_R + numberRAM$;
15: $\quad\quad numRead_D \leftarrow numRead_D + |L_i^{'}| + |L_{i+1}^{'}| - numberRAM$;
16: $\quad\quad i \leftarrow i + 1$;
17: $\quad$ **end while**
18: **end while**
19: $cost_{insert} \leftarrow \frac{numRead_R \cdot R_R(B) + numWrite_R \cdot W_R(B)}{numInsert} + \frac{O(MapLen)}{numInsert}$;
20: $cost_{insert} \leftarrow \frac{numRead_D \cdot R_D(B) + numWrite_D \cdot W_D(B)}{numInsert} + cost_{insert}$; **return** $cost_{insert}$;

Line 2 initializes four variables which denote the number of reads and writes in main and secondary memory, respectively. $|L_i^{'}|$ in line 7 denotes the current number of entries in L_i, and it is set to be zero before insertion. Line 19 calculates the overhead of accessing main memory and synchronization with secondary memory which is in proportion to the mapping length. Line 20 calculates the whole overhead of insertion. Then we use $cost = W_i \cdot cost_{insert} + W_s \cdot cost_{search}$, $W_i + W_s = 1$ to estimate comprehensive overhead and choose the optimal k under different size of available main memory.

Algorithm 3 $mode\_set(IndexHead, MapLen)$

Require: $IndexHead$:structure which stores important parameters of the index; $MapLen$: the length of memory mapping;
Ensure: $TURE$ for success, $FALSE$ for failure;
1: $f \leftarrow f_{RAM}$, choose the optimal k according to environment;
2: Compute the new index file size $|L|$,update it and initialize the index head;
3: $IndexHead \cdot base \leftarrow mmap(fp, MapLen)$;
4: $read(IndexHead, fp, 0)$;
5: **if** $|L| < MapLen$ **then**
6: **for** each level in Tide-tree **do**
7: $LAT[i].u_{RAM} \leftarrow 1$;
8: **end for**
9: **else**
10: $f \leftarrow f_{Disk}$, choose optimal k according to environment;
11: Compute index file size $|L|$ and initialize index head;
12: **for** $i \leftarrow 1$ *to* $l-1$ **do**
13: **if** $MapLen > \sum_{j=0}^{i} IndexHead.level[j].length$ **then**
14: $LAT[i].u_{RAM} \leftarrow 1$;
15: **else**
16: $LAT[i].u_{RAM} \leftarrow \frac{MapLen - LAT[i].O_{start}}{LAT[i].O_{end} - LAT[i].O_{start}}$;
17: **if** $LAT[i].u_{RAM} < 0$ **then**
18: $LAT[i].u_{RAM} \leftarrow 0$;
19: **end if**
20: **end if**
21: **end for**
22: **end ifreturn** $TURE$;

3.4.2 Mode setting algorithm

For the index structure being built or loaded, we need to calculate the node size, the level ratio and the mode of each level as explained in last section first. Then all the information needs to be updated to *index head*. Algorithm 3 shows the details of mode setting.

We set the size of available continuous space in main memory as the mapping length-$MapLen$. By default, we employ the binary search to get the accurate length by setting different length and mapping frequently. If the users just do not want to occupy too much main memory, the mapping length can be manually set. In lines 2-9 we set the node size as if the data is in main memory, in order to see whether main memory can accommodate the whole tree and set the mode of each level as *RAM-mode*. In lines 10-22 we set the node size as same to scenario in secondary memory, and we calculate each level's mode by the equation in line 16. The f_{RAM} and f_{Disk} are evaluated in Section 5.2.

4 Key operations of tide-tree

This section mainly introduces important operations of Tide-tree, including index creation, merge operation, index load, index rebuild and range search, etc.

4.1 Index Creation

The index creation is composed of index file's creation, mapping to main memory, building index in main and secondary memory, and flushing the data from main to secondary memory eventually. The difference compared with FD-tree is that we utilize more available main memory to improve the efficiency but still maintain the index to disk completely.

Algorithm 4 $tide\_create\ (|L|, n, KEY)$

Require: $|L|$: the initial size of file,n:the number of indexing entries, KEY : the entries set;
Ensure: $root$:the offset of root node in index;

1: $fd \leftarrow create\_file(|L|)$;
2: $tide\_load(fd)$;
3: **for** each $key \in KEY$ **do**
4: $\quad tide\_merge(LAT)$;
5: $\quad tide\_insert(key, LAT)$;
6: **end for**
7: $root \leftarrow LAT\,[0]\,.head$;
8: $msync\left(\sum_{i=0}^{l-1} |L_i| \cdot |e| \cdot L_i.u_{RAM}, MS\_SYNC\right)$; **return** $root$;

In every insertion, we need to see whether to merge adjacent levels by checking the offset of each level. The line 8 of Algorithm 4 applies the asynchronous refreshing way (MS_SYNC denotes the mode) to make the data durable in secondary storage, this will save much time and have small delay compared to reads and writes of file.

4.2 Merge operation

The merge operation reduces the overhead of writing to disk by transforming the random writes to sequential writes. By memory mapping technology, we can write larger blocks into secondary memory and do not need the buffer pool anymore, the throughput will be much higher. Before the insertion, we need to check each level whether reaching its max capacity from root to leaf.

Algorithm 5 $tide\_merge\ (LAT)$

Require: LAT:the level address table;
Ensure: $TURE$ for success, $FALSE$ for failure;

1: **for** each level i in LAT **do**
2: $\quad$ **if** $LAT[i].O_{level} \geq LAT[i].O_{end} - LAT[i].O_{start}$ **then**
3: $\quad\quad merge\_level(L_i, L_{i+1})$;
4: $\quad\quad tide\_rebuild(L_{i+1})$;
5: $\quad$ **end if**
6: **end for**
7: $msync\left(\sum_{i=0}^{l-1} |L_i| \cdot |e| \cdot L_i.u_{RAM}, MS\_SYNC\right)$; **return** $TURE$;

The line 2 means that the level has exceeded the prescriptive capacity and merges with next level, then rebuilds the upper levels at line 3. The rebuilding operation fetches the first entry of each node in L_{i+1}, initiates it as external fence pointing to itself, and insert to L_i, repeats this operation until getting to the root of tree.

4.3 Index loading

This operation can load the index data to main memory from secondary memory partly or entirely. It will save lots of computing resource because reads always surpass writes, especially for the big data and SSD scenario, which applies the erase-before-write mechanism, the latency of erase operations is far higher than reads or writes. Additionally, the frequent write will decrease the life of device. As a result, this causes inferior write, especially random write performance. So this operation is very helpful in the presence of errors(e.g. power failures) and can be applied to the SSD environment for omitting writes.

Algorithm 6 $tide\_load\ (fp)$

Require: fp: the file pointer of index file;
Ensure: $root$: the offset of root in index;
1: $mode\_set(LAT)$;
2: $root \leftarrow IndexHead.root + IndexHead.base$;
3: $node \leftarrow read\_block(root, B)$;
4: $EnQueue(q, node)$;
5: **while** $node! = NULL\&GetOffset(node) < IndexHead.MapLen$ **do**
6: $DeQueue(q, node)$;
7: **for** each child $cnode$ **of** $node$ **do**
8: $EnQueue(q, cnode)$;
9: **end for**
10: **end whilereturn** $root$;

Lines 2-10 travel the mapping node of index tree for reading the data to main memory actually, because the mapping operation just builds an one-to-one relation between main and secondary memory. However, the index data is still located in disk, there will be a missing page interruption only by accessing the data in secondary memory, the data will be loaded into main memory really. After the load operation, the upper levels will be located in main memory.

4.4 Range search

The key point of search operation is that the accessed nodes from root to leaf may have different modes. In order to guarantee universality, we will employ Algorithm 1 in Section 3.4.1 to access the nodes. Another key point is that we need to analyze and judge the types of entries, because the target keys may locate in *Head tree* or lower levels, the search will stop before reaching the leaf and be more effective. Our point search is same as FD-tree, and our emphasis is the range search.

Since there is no normal entry in internal levels when the tree is not in dirty state, range search can be efficient B^+-tree; otherwise we need to travel level by level. The procedure of range search is described in Algorithm 7.

Algorithm 7 $tide\_range\_search\ (low,\ high)$

Require: $low, high$: the boundaries of range;
Ensure: R: the result set;
1: $R \leftarrow \emptyset$;
2: **if** $!Indexhead \cdot M_{dir}$ **then**
3: **for** $i \leftarrow 0\ to\ l-2$ **do**
4: $offset_1 \leftarrow get\_offset\_level(i, low)$;
5: $offset_2 \leftarrow get\_offset\_level(i, high)$;
6: **if** $!L_i \cdot M_{dir}$ **then**
7: $i \leftarrow i+1$;
8: **else**
9: $R \leftarrow R \cup search\_level(offset_1, offset_2, i)$;
10: **end if**
11: **end for**
12: **end if**
13: $offset_1 \leftarrow get\_offset\_level(l-1, low)$;
14: $offset_2 \leftarrow get\_offset\_level(l-1, high)$;
15: $R \leftarrow R \cup search\_level(offset_1, offset_2, l-1)$; **return** R;

Lines 3-12 cater to the case that there exists index entries beyond the leaf level, so we need to find them and add the searched entries into result set. Once occurring a collision, which means there exists a filter entry and a normal entry in the same time, we will not add it to the result set and delete the corresponding normal entry. Lines 13-16 travel the leaf level.

We set the selectivity of range search as s, so the number of target index entries is $N \cdot s$. The best case of our scheme is that all entries lie in leaf level, so the overhead is $\frac{N \cdot s}{f} \cdot R\,(B)$, while the origin scheme will cost $\frac{N \cdot s}{f} \cdot R\,(B) \cdot \sum_{i=1}^{l} \frac{1}{f^{i-1}}$. In this case, we can save the biggest overhead, while the smallest is 0 when all levels contain index entries. So the mean overhead reduced can be denoted as $\frac{N \cdot s}{2f} \cdot R\,(B) \cdot \sum_{i=2}^{l} \frac{1}{f^{i-1}}$. The overhead of point search is calculated as (6), which means the overhead of point search can be denoted as $O\left(\log_k n\right)$.

4.5 Rebuild operation

When the running devices change such as data migration [6], or the workloads change such as various applications are coming, we need to evaluate the size of node and level ratio again, and rebuild the index to fulfill optimal performance. The rebuild operation is mainly executed before index loading under storage sense. The details of rebuild operation are described as Algorithm 8.

Algorithm 8 $tide\_rebuild(LAT)$

Require: LAT: the level address table;
Ensure: $TURE$ for success, $FALSE$ for failure;
1: $i \leftarrow 0, j \leftarrow 0$;
2: **if** $IndexHead.M_{dir}$ **then**
3: **for** each level L_i in Tide-tree **do**
4: **if** $L_i.M_{dir}$ **then**
5: $j \leftarrow i + 1$;
6: **while** $!L_j.M_{dir}$ **do**
7: $j \leftarrow j + 1$;
8: **end while**
9: **end if**
10: $merge\_level\left(L_i, L_j\right)$;
11: $i \leftarrow j$;
12: **end for**
13: $IndexHead.M_{dir} \leftarrow false$;
14: **end if**
15: Compute the new node size and level ratio;
16: Change the node size of leaf level;
17: **for** each level L_i in Tide-tree **do**
18: Choose the first entry of new node to rebuild the upper level;
19: **end forreturn** $TURE$;

The line 15 utilizes the framework of Section 3.4.1 to calculate the optimal node size and level ratio. The rebuild operation not only makes Tide-tree adaptive to different environment, but also turns the dirty state to be false. The query performance will be more stable and efficient, and the range search will be the most efficient.

5 Experimental evaluation

We implement a prototype of Tide-tree with C language under our own main memory database, and compare against famous indexing schemes from different angles. To the best of our knowledge, there are no indices proposed for RAM/Disk-based hybrid storage systems, so we choose those representative main memory and disk-based indices to compare. We first show the experiment environment and dataset, and then conduct four groups of experiments under different storage device and scenario. We start from following points: (1) the evaluation of node size and level ratio; (2) the performance compared with main memory indices when the RAM is adequate; (3) the performance compared with RAM/Disk-based indices when the RAM is inadequate; (4) the footprint compared with above schemes.

5.1 Experiment setup

The hardware environment of experiments conducted below is shown as follows: Intel(R) Core(TM) i3-2120 CPU 3.3GHz 4 core processor, Kingston DDR3 1333 8G RAM, the

Table 4 Parameter setting

Entry	Value
Page size(SSD)	4KB
Cache line size	64B
t	2
$\|IH\|$	128B

secondary storage devices include OCZ SATA SSD(120G), Intel 530 SSD(120G) and WDC HDD(500G). The software environment is: Ubuntu 12.04 64bit operating system, GCC 4.6.3 compiler. Table 4 shows the parameters in the experiment.

For simplicity, we set various mapping lengths to simulate different environment. We use $Tide - tree(i)$, $i \in [0, 1]$ to represent all scenarios of Tide-tree, where i denotes the ratio between the mapping length and size of whole index file, we call it loading ratio. Tide-tree is a main memory index when $i = 1$, and it is totally a disk-based index when $i = 0$; otherwise it is a RAM/Disk-based hybrid index for most scenarios. By this means, we can simulate all scenarios without changing too many main memory devices with different sizes. We have used our synthetic data sets and query for a better control on their characteristics, the synthetic datasets includes the sorted and unsorted integers from 1-10,000,000.

5.2 Node size and level ratio

Figure 9 shows the search performance with the varying of node sizes, we just take the search overhead into account. To eliminate the influence of level ratio, we set $k = f - 2$ and build the tree from bottom to top with sorted keys. Once determining the optimal f,

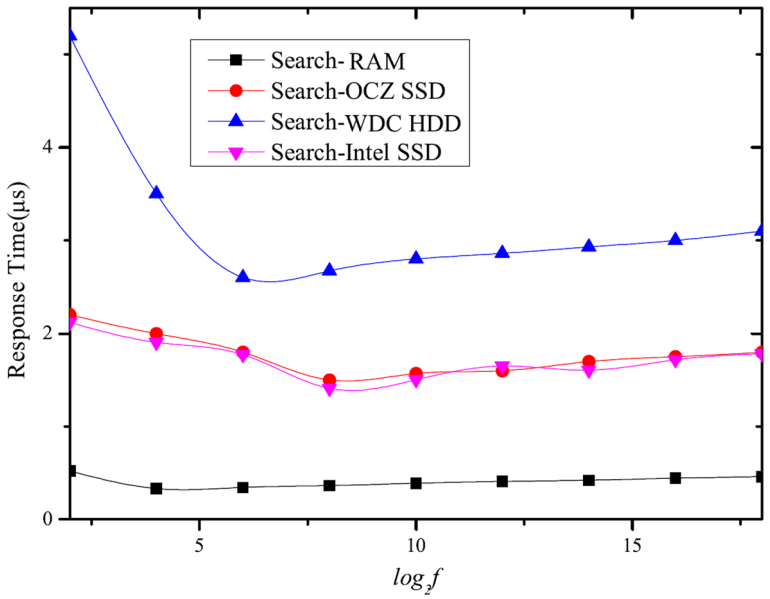

Figure 9 Performance with varying node size

we can turn the k according to the environment. The horizontal axis shows the logarithm of f, all these cases have the same trend, decreases first and rises monotonically. The only difference is that the curve of running in main memory gets earlier peak, the HDD gets latter and the two SSDs gets the latest. This is due to the different access efficiency of the running environment. So when the whole tree is located in main memory, we set the size of node as the peak value, denoted as f_{RAM}; otherwise we set it as page size of SSD, denoted as f_{Disk}. Because the minimal granularity of reads and writes is a page in SSD, we choose the same size for HDD to unify.

Figure 10 shows the insertion and search performance with varying of level ratio by inserting thousands of keys. The horizontal axis represents the value of k, and we can see that the curves of insertion rise monotonously. The bigger the level ratio is, the bigger the overhead cost by the insertion is. If we just taking the insertion efficiency into account, it is a good choice to choose the smallest k, but the footprint of each level is almost equal, which means the footprint is very large. On the other hand, the smaller the k is, the bigger the height is, so the overhead of search will be large. This is verified by the two decreasing curves of search, that is why we use weighting factor to choose optimal level ratio. Since the two SSDs get similar performance, so we just list the result of OCZ SSD in the next.

5.3 Performance comparison on main memory

In this section we mainly evaluate Tide-tree's performance running in main memory. We choose two representative main memory indices to compare. Since deletion and updating operation are all composed of insertion and search, we will only evaluate the insertion and point search.

Figure 11 compares the performance between Tide-tree and other famous main memory indices, including the CSB$^+$-tree, DPT [19], which achieve well in main memory both using cache technology. The horizontal axis shows the logarithm of scale size, the vertical axis

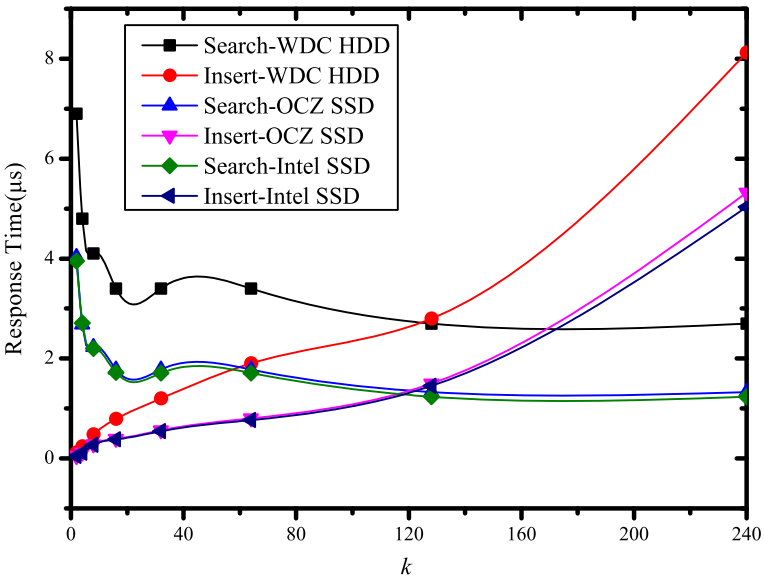

Figure 10 Performance with varying level ratio

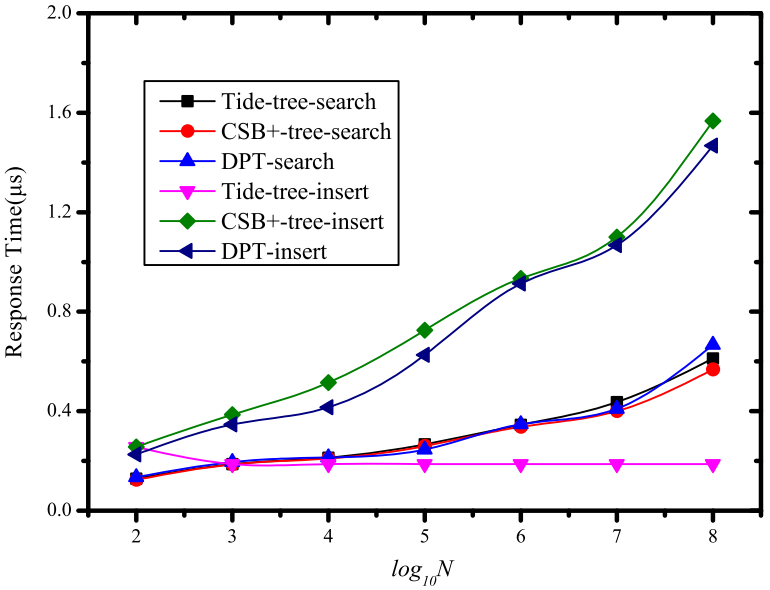

Figure 11 Performance comparison of main memory indices

shows the response time of different operations. We find that the search and insertion of proposed scheme do not degrade too much for loading from the disk, that is because we use an improved pointer swizzling technology to overcome the shortcoming of translation table. We can also find that the overhead of insertion is much less than other indices, because we just insert the keys into *Head tree* first, the insertion is most stable under all scenarios, it

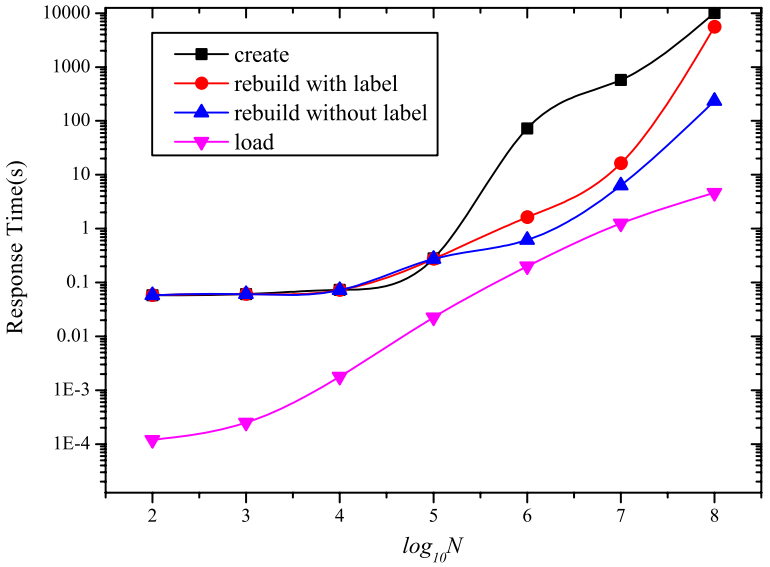

Figure 12 Comparison between four kinds of creations

has a low latency which can be used in those fields which require high performance of data updating.

Figure 12 shows the performance of index creation, including creation from data table, rebuilding from index file and loading from index file. We can see that the index load operation has the best performance comparing with other three operations and wins a couple of orders of magnitude smaller than creation from data table. The figure also shows that the rebuilding with label saves much time compared with rebuilding without label. This is accomplished through the dirty identifiers of whole tree and each level in *index head*, it will not cost much space and much time to access, but can jump directly to the level which contains target entries.

We can draw the conclusion that Tide-tree has a good performance on main memory while it has a copy on disk, which makes it durable and reusable, these characters will be helpful in the field which needs high instantaneity when system crashes. Furthermore, Tide-tree does not degrade too much in query when the RAM is adequate, it is still a competitive choice.

5.4 Performance comparison on RAM/Disk

This section evaluates the performance of Tide-tree when it acts as a RAM/Disk-based hybrid index in terms of effectiveness and efficiency. Since the nodes accessed by a query are located in or out, the stability of query is very important. The horizontal axis in Figures 13 and 14 shows the number of queries. We can see that the search do not fluctuates too much when the loading ratio is small. Only when the loading ratio get a threshold, the tree begin to fluctuate in a small range. When the ratio equals to 1, the performance will be most stable and efficient.

We choose the popular disk-based indices, FD-tree and B^+-tree to compare the performance. Figures 15 and 16 shows the inserting and searching overhead of index file,

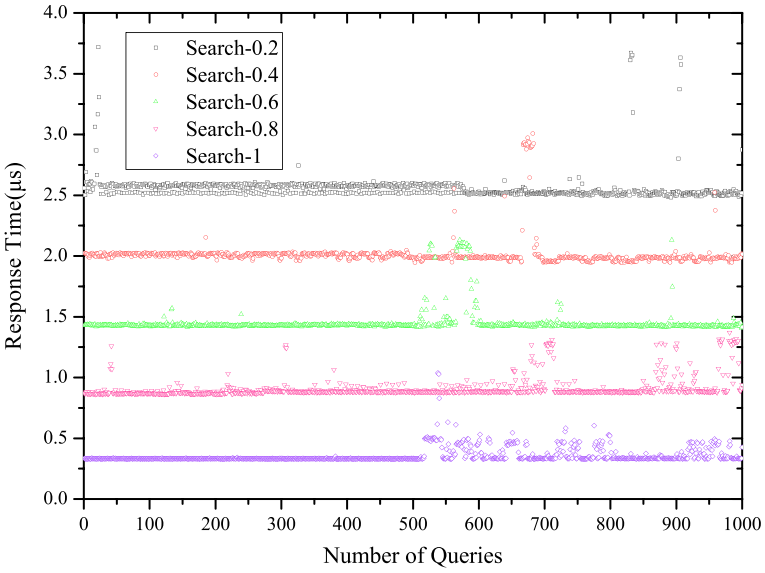

Figure 13 The search performance under different loading ratio in HDD

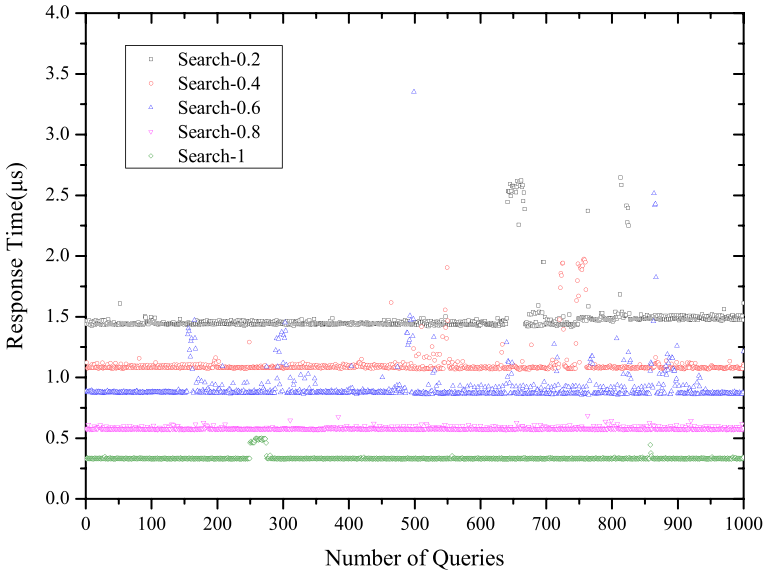

Figure 14 The search performance under different loading ratio in SSD

completed by inserting many keys into a same size index so leads to merge operation and flushing to disk. We can see that our scheme is more efficient than FD-tree and B^+-tree when the loading ratio is high. When the ratio is quite small, they are very close. This is because

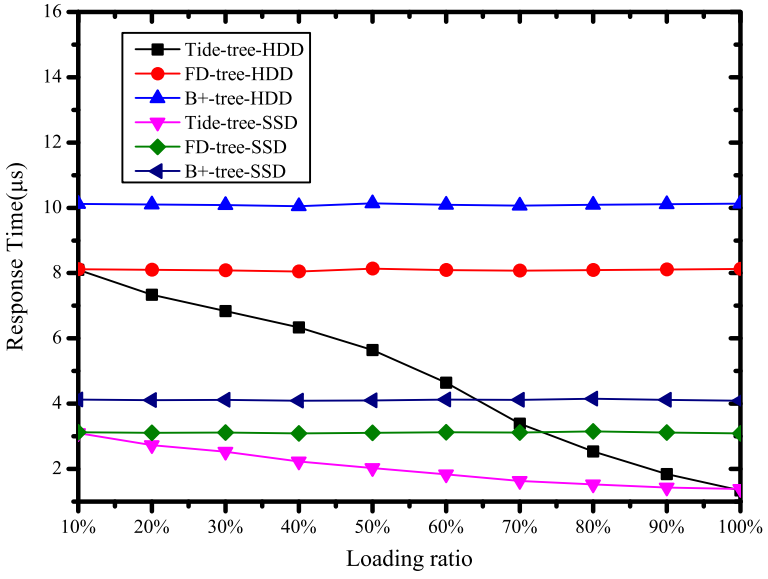

Figure 15 Inserting overhead vs loading ratio

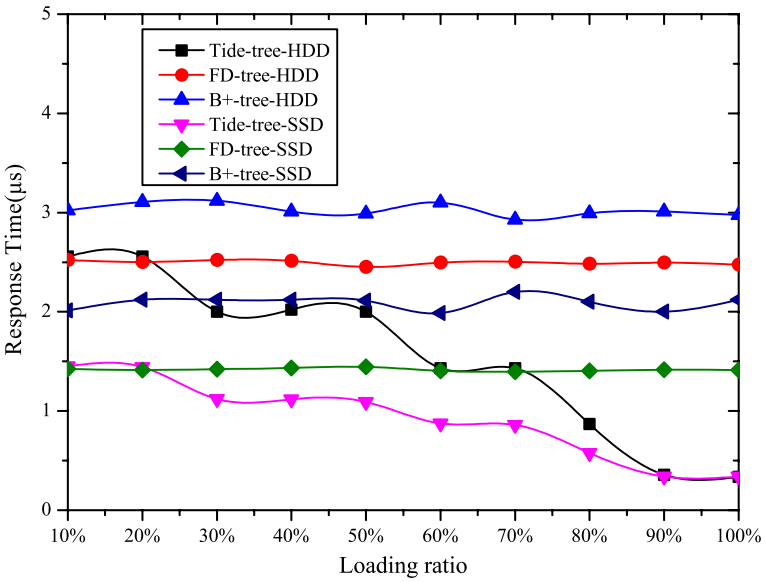

Figure 16 Searching overhead vs loading ratio

the larger the ratio is, the more area is updated by memory mapping technology, the bigger block flushes to disk once and the bigger the bandwidth is. This verities our thought in Section 3.1. So the scheme is most efficient when the ratio is 1. In other words, Tide-tree

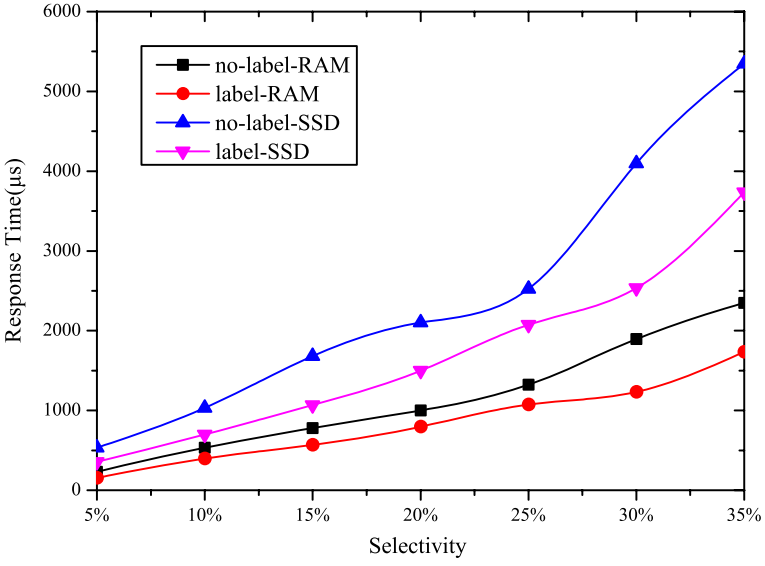

Figure 17 Range search under different selectivity

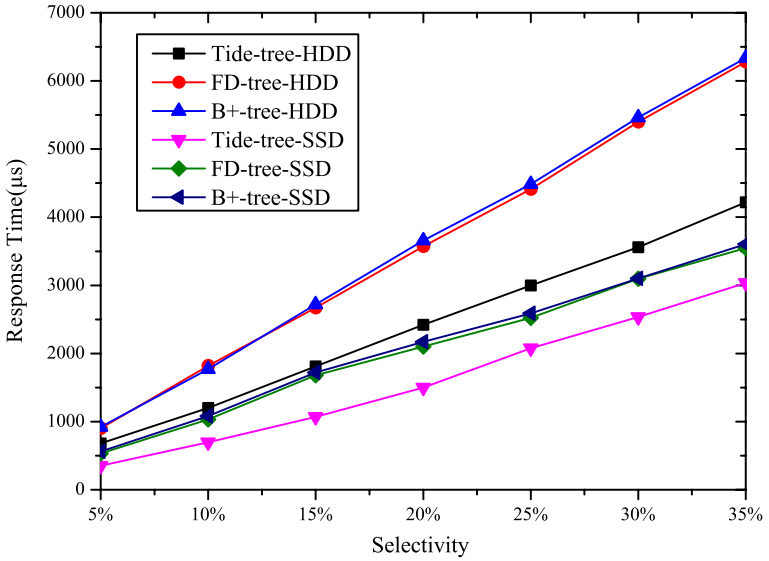

Figure 18 Range search overhead vs selectivity

acts as a main memory index at all. Since the deletion in Tide-tree and FD-tree are composed of insertion and search both, so we will not compare the deletion here.

Figures 17 and 18 shows the performance of range search. We compare the label-based range search with normal range search first. The horizontal axis of Figure 17 shows the selectivity which represents the size ratio between result set and whole index. As the selectivity increases, the overhead increases linearly. The normal range search is much higher than the label-based one because it needs to search those levels which do not contain any target entries. It wastes much time in reading blocks into main memory from secondary memory. In Figure 18, we set the loading ratio of Tide-tree as 10 %, and it shows that Tide-tree outperforms the FD-tree and B^+-tree both when employing the label-based algorithm.

In summary, Tide-tree has an equal performance to the FD-tree when we do not load index into main memory. However, when we utilise main memory partly or entirely to process all kinds of query, the performance will be higher than FD-tree and B^+-tree both. This is because main memory outperforms secondary memory in terms of access speed, while we have overcome the volatile issue of main memory index to some extent, so we feel free to use the main memory. The main memory is adequate in most application scenarios, except in the big data scenarios. But we can turn our scheme to utilize available secondary memory, which is another key character of Tide-tree.

5.5 Memory footprint

This section mainly compares the footprint of whole index with above compared schemes, such as FD-tree, B^+-tree, CSB^+-tree. We still set different number of keys to observe the difference in Figure 19. In Figure 20, we set the number of keys as 10^7 to see the footprints of main and secondary memory.

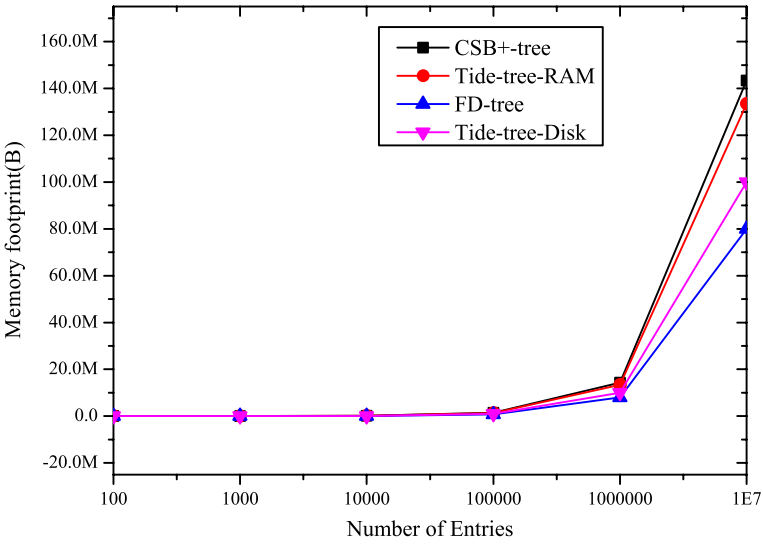

Figure 19 Memory footprint vs number of entries

We can see that when acting as a main memory index, Tide-tree has a bigger footprint than running in secondary memory, because the former has a smaller node and has more fences in internal level. We make our scheme more adaptive without increasing footprint too much, because main memory is always smaller and more expensive than SSD and HDD.

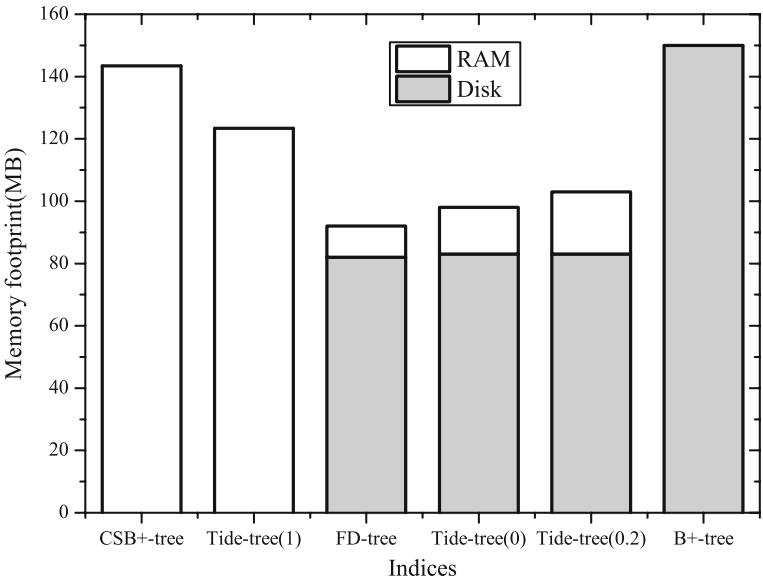

Figure 20 Memory footprint of different indices

6 Conclusions

In this paper, we propose a self-tuning indexing scheme for RAM/Disk-based hybrid storage system, which can be adaptive to diverse environments by an effective self-tuning algorithm. Based on FD-tree, we utilize available main memory and increase the accessing block size to improve the throughout by memory mapping technology. Furthermore, we add three efficient operations and improve the range search operation beyond FD-tree, the index load operation can save much computing resource compared with the index creation from data table. With *index head* and *LAT*, we improve the pointer swizzling technology, which abandons translation table and saves much time and space. Tide-tree has good transportability, reusability and is appropriate for those environment with heterogeneous computing resources.

In the big data processing scenario, single computing node will not be able to process data efficiently, so we consider to employ proposed scheme to the Cloud index in the future, like the Hadoop or Spark, to optimize it for cluster computing framework.

Acknowledgments This work is partially supported by the Australian Research Council's Discovery Projects Scheme (DP170102726), the National Natural Foundation of China under Grant No. 91646204, 61373015, 61300052, 41301047, 71322104, the Funding of Jiangsu Innovation Program for Graduate Education under Grant No.SJZZ_0043, and National Center for International Joint Research on E-Business Information Processing (2013B01035).

References

1. Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., Singh, S.: Lazy-adaptive tree: An optimized index structure for flash devices. Proceedings of the VLDB Endowment **2**(1), 361–372 (2009)
2. Athanassoulis, M., Ailamaki, A.: Bf-tree: approximate tree indexing. Proceedings of the VLDB Endowment **7**(14), 1881–1892 (2014)
3. Boehm, M., Schlegel, B., Volk, P.B., Fischer, U., Habich, D., Lehner, W.: Efficient in-memory indexing with generalized prefix trees. In: BTW, vol. 180, pp. 227–246 (2011)
4. Chaudhuri, S., Weikum, G.: Rethinking database system architecture: Towards a self-tuning risc-style database system. In: VLDB, pp. 1–10. Citeseer (2000)
5. Comer, D.: Ubiquitous b-tree. ACM Comput. Surv. (CSUR) **11**(2), 121–137 (1979)
6. Das, S., Nishimura, S., Agrawal, D., El Abbadi, A.: Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. Proceedings of the VLDB Endowment **4**(8), 494–505 (2011)
7. Dewitt, S.J.W.D.J.: A performance study of alternative object faulting and pointer swizzling strategies. In: Proceedings 18th Int. Conf. Very Large Data Bases, Vancouver, BC, Canada (1992)
8. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: Sql server's memory-optimized oltp engine. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1243–1254. ACM (2013)
9. Fu, Z., Ren, K., Shu, J., Sun, X., Huang, F.: Enabling personalized search over encrypted outsourced data with efficiency improvement. IEEE Trans. Parallel Distrib. Syst. **27**(9), 2546–2559 (2016)
10. Fu, Z., Wu, X., Guan, C., Sun, X., Ren, K.: Towards efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. IEEE Transactions on Information Forensics and Security, doi:10.1109/TIFS.2016.2596138 (2016)
11. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database system implementation, vol. 654. Prentice Hall Upper Saddle River, NJ (2000)
12. Graefe, G.: Modern b-tree techniques. Foundations and Trends in Databases **3**(4), 203–402 (2011)
13. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 371–381. ACM (2010)
14. Graefe, G., Volos, H., Kimura, H., Kuno, H., Tucek, J., Lillibridge, M., Veitch, A.: In-memory performance for big data. Proceedings of the VLDB Endowment **8**(1), 37–48 (2014)

15. Halim, F., Idreos, S., Karras, P., Yap, R.H.: Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. Proceedings of the VLDB Endowment **5**(6), 502–513 (2012)
16. Idreos, S., Kersten, M.L., Manegold, S., et al.: Database cracking. In: CIDR, vol. 3, pp. 1–8 (2007)
17. Jin, P., Yang, P., Yue, L.: Optimizing b+-tree for hybrid storage systems. Distributed and Parallel Databases **33**(3), 449–475 (2015)
18. Jørgensen, M.V., Rasmussen, R.B., Šaltenis, S., Schjønning, C.: Fb-tree: a b+-tree for flash-based ssds. In: Proceedings of the 15th Symposium on International Database Engineering & Applications, pp. 34–42. ACM (2011)
19. Kissinger, T., Schlegel, B., Boehm, M., Habich, D., Lehner, W.: A high-throughput in-memory index, durable on flash-based ssd: insights into the winning solution of the sigmod programming contest 2011. ACM SIGMOD Record **41**(3), 44–50 (2012)
20. Lahiri, T., Neimat, M.A., Folkman, S.: Oracle timesten: An in-memory database for enterprise applications. IEEE Data Eng. Bull. **36**(2), 6–13 (2013)
21. Lee, H.S., Lee, D.H.: An efficient index buffer management scheme for implementing a b-tree on nand flash memory. Data Knowl. Eng. **69**(9), 901–916 (2010)
22. Lehman, T.J., Carey, M.J.: A study of index structures for main memory database management systems. In: Proceedings VLDB (1986)
23. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 38–49. IEEE (2013)
24. Li, Y., He, B., Luo, Q., Yi, K.: Tree indexing on flash disks. In: 2009 IEEE 25th International Conference on Data Engineering, pp. 1303–1306. IEEE (2009)
25. Li, Y., He, B., Yang, R.J., Luo, Q., Yi, K.: Tree indexing on solid state drives. Proceedings of the VLDB Endowment **3**(1-2), 1195–1206 (2010)
26. Lin, Z., Kahng, M., Sabrin, K.M., Chau, D.H.P., Lee, H., Kang, U.: Mmap: Fast billion-scale graph computation on a pc via memory mapping. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 159–164. IEEE (2014)
27. Long, X., Suel, T.: Three-level caching for efficient query processing in large web search engines. World Wide Web **9**(4), 369–395 (2006)
28. Mullin, J.K.: A second look at bloom filters. Commun. ACM **26**(8), 570–571 (1983)
29. Nath, S., Kansal, A.: Flashdb: dynamic self-tuning database for nand flash. In: Proceedings of the 6th international conference on Information processing in sensor networks, pp. 410–419. ACM (2007)
30. Peng, P., Zou, L., Chen, L., Lin, X., Zhao, D.: Answering subgraph queries over massive disk resident graphs. World Wide Web **19**(3), 417–448 (2016)
31. Rao, J., Ross, K.A.: Making b+-trees cache conscious in main memory. In: ACM SIGMOD Record, vol. 29, pp. 475–486. ACM (2000)
32. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), pp. 1–10. IEEE (2010)
33. Song, S., Chen, L.: Indexing dataspaces with partitions. World wide web **16**(2), 141–170 (2013)
34. WANG, S., QIN, X., SHEN, Y., LI, B., SHI, W.: Research on durable csb+-tree indexing technology. Journal of Frontiers of Computer Science and Technology **2**, 005 (2015)
35. Wu, C.H., Kuo, T.W., Chang, L.P.: An efficient b-tree layer implementation for flash-memory storage systems. ACM Trans. Embed. Comput. Syst. (TECS) **6**(3), 19 (2007)
36. Xia, Z., Wang, X., Sun, X., Wang, Q.: A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. IEEE Trans. Parallel Distrib. Syst. **27**(2), 340–352 (2016)
37. Yang, C., Jin, P., Yue, L., Yang, P.: Efficient buffer management for tree indexes on solid state drives. Int. J. Parallel Prog. **44**(1), 5–25 (2016)
38. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 2–2. USENIX Association (2012)
39. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**, 10–10 (2010)