Using Evolving Storage Structures for Data Storage

Syed Saif ur Rahman
Department of Technical and Business Information Systems
Faculty of Computer Science
Otto-von-Guericke University of Magdeburg, Germany
srahman@ovgu.de

ABSTRACT

Different data storage structures suit different data management scenarios, therefore, a universal data storage structure is not possible. Furthermore, each data storage structure has different resource consumption because of their unique execution complexity. Self-tuning data management systems need a mechanism to select appropriate data storage structure and to adapt it with changing data management requirements. We propose Evolving Storage Structures for self-tuning data management system that supports customization and adaptation with changing data management needs.

Categories and Subject Descriptors

H.2 [**Database Management**]: Database architectures, Physical Design, Access methods

General Terms

Design, Performance

Keywords

Cellular DBMS Architecture, Evolution, Storage Structures, Self-tuning

1. MOTIVATION AND CONTRIBUTION

Existing data management solutions are complex [3]. They are full of functionalities that are added over time. Furthermore, most of functionalities are tightly integrated with each other. It is difficult to customize an existing data management solution through removing unused functionalities. We argue that unused functionalities in data management solutions cause overheads that contribute to more resource consumption, which in turn reduces performance. We evaluated an existing data management solution, i.e., BerkeleyDB [9,10] with our custom micro benchmark to prove our argument. Results from Saake et al. in [12] also strengthens our argument by presenting, how existing database systems can be downsized and in turn can give better performance. According to our evaluation and results from Saake et al. in [12], we identified the need for customization of data management solutions at fine-granularity.

Different storage structures in existing data management solutions have different execution complexity. By execution complexity, we mean the memory footprint, function calls, branches and mispredictions, cache references and misses, etc., by a storage structure during data management operations. For clarity, we classified storage structure complexity into three categories, i.e., simple, average, and complex. We argue that simple storage structures are appropriate for small data management tasks and consume fewer resources in comparison with complex storage structures. As the data size grows, average complexity storage structures perform better with appropriate resource consumption in comparison with simple and complex storage structures. For large data management tasks, complex storage structures are the appropriate solution. To be more concrete with our example, we use a sample classification in Table 1, which uses the results provided by Lehman and Carey in [7].

Furthermore, it can be observed from Table 1 that different storage structures are suitable for different workloads and data sizes. Each storage structure exposes different merits and demerits. We cannot find a universal storage structure that can perform optimally for all data sizes and workloads with appropriate resource consumption. To prove our argument, we evaluated different storage structures over different data sizes with similar workload. Details about evaluation and results are provided in Section 4.

In real world scenarios, we face diversified data management needs. Different storage structures are appropriate for different scenarios, which require extensive tuning. Self-tuning data management solutions attempt to solve this problem automatically with minimum human intervention, however, there exist the need for a mechanism that should facilitate appropriate storage structure selection and tuning. We propose Evolving Storage Structures as an alternative solution, which supports selection of an appropriate storage structure through customization and supports change in storage structure with change in data management needs.

During query processing, different DBMS operations operate over a source data in logical order to generate desired results. Especially in column-oriented DBMS, such as the MonetDB[1] with column-at-a-time MIL execution model, these operations generate intermediate results that vary in size and number of attributes [2]. DBMS may materialize

[1] "MonetDB", `http://monetdb.cwi.nl/`

FIT '10, December 21-23, 2010, Islamabad, Pakistan.

Storage structure	Complexity class	Data size class	Benefits	Problems
Sorted Array	Simple	Small	Read optimized (Good data reference locality) Cache and space efficient	Write/Update (Requires rearrangement)
Heap Array	Simple	Small	Write optimized	Search time (Poor data reference locality) Complete scan for duplicates
Sorted List	Average	Medium	Read optimized	Write/Update(Requires rearrangement)
Heap List	Average	Medium	Write optimized	Search time (Poor data reference locality) Complete scan for duplicates
Hash Table	Average/Complex	Medium/Large	Write optimized Memory efficient Unordered data access	High space overhead (For dynamic hash tables) It does not preserve order Complete bucket scan for duplicates Range queries
B+-Tree	Average/Complex	Medium/Large	Suited for disk use Fast search and update More cache conscious Range queries efficient	Not good for memory

Table 1: Storage structures classification.

these results to reduce the query processing time. For example, MonetDB uses the binary association table (BAT) to store intermediate results. We argue that we cannot determine the most optimal storage structure for these intermediate results, and we propose the use of evolving storage structures for intermediate results materialization.

In this paper, Section 2 explains concepts of evolving storage structures. Section 3 introduces the concept of the evolution path and proposes schemes to alter the inefficient evolution path. Section 4 discusses the evaluation of presented arguments and approaches. Section 5 documents the related work. Section 6 concludes the paper with few hints for future work.

2. EVOLVING STORAGE STRUCTURES

Evolving storage structures are hierarchically-organized storage structures. By a hierarchical organization, we mean a composition of similar or different storage structures in a hierarchy as depicted in Figure 1. Evolving storage structure is initialized as an atomic, autonomic, and customized minimal storage structure, e.g., Page or Sorted List of pages. By customization, we mean the selection of an appropriate storage structure, i.e., either page saves data as a sorted array or a heap array. Each storage structure stores key/value pair of data. By atomicity, we mean that storage structure exposes an API, and it is capable of storing data independently, i.e., a single page exposes an API, and it can be used to store data independently. By autonomy, we mean the capability to monitor, diagnose, and tune the storage structure. To hide the complexity of differences in storage structures, evolving storage structures expose consistent API across the hierarchy.

Evolving storage structures impose storage capacity constraint on each storage structure to ensure that each storage structure has predictable performance for data management tasks. As long as data can be stored within a single page, it is used for data storage, but when data size reaches the limit of the page storage capacity. Storage structure is evolved, i.e., for example, storage structure A (i.e., a page) evolves into new storage structure B such that A becomes an integral unit of B. We call this process "Evolution". A new storage structure (e.g., Sorted List or Hash table or B-Tree of multiple pages) is selected based on known heuristics and the monitoring and analysis of workload performed during the data storage in a page. Storage structure hierarchy should also consider the hardware hierarchy for next optimal storage structure selection. As depicted in Figure 1, the initial smallest and simplest structure is optimized for cache, than one above is optimized for memory, and than one above is optimized for disk drive. A sample scenario of storage structure evolution in evolving storage structures is shown in Figure 3.

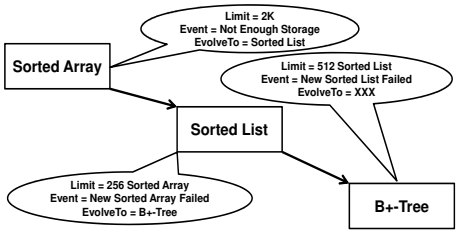

Figure 3: Sample hierarchically-organized storage structure evolution scenario.

During evolution, we also recommend an optional mechanism to increase the efficiency of evolving storage structures, i.e., splitting and compaction. Splitting and compaction means to break the storage structure into suitable small-sized structures, such that storage structure A is distributed into n compact structures A'. The need for a splitting arises when we use a large structure initially to reduce resource consumption or to increase the reliability of workload monitoring decision. For example, we initialize an evolving storage structure as a page with page size of 8KB. Then, when we evolve this page into a higher-level structure (e.g., Sorted List), it is recommended to split this page into multiple small size pages with equal data distribution among them.

Evolving storage structures are highly customizable and reconfigurable. Figure 2 demonstrates the customizability and reconfigurability of a sample evolving storage structure. It shows that at each level of hierarchy, a storage structure can be customized. It also shows that at each evolution event, it is decided what will be the next storage structure type in a hierarchy? Each new level storage structure is composed of multiple lower level storage structures.

3. EVOLUTION PATH

Evolution path is the mechanism to define how storage structures evolve from smallest simple storage structure into large complex storage structures. It consists of many storage structure/mutation rules' entries. Each storage structure can have multiple mutation rules mapped to it. Each mutation rule is composed of three information elements, i.e., Event, Heredity-based-selection, and Mutation. An event identifies, when a mutation rule should be executed. Heredity-based-selection identifies precisely, when evolution should occur based on the heredity information gathered for current storage structure. Mutation defines actions that should

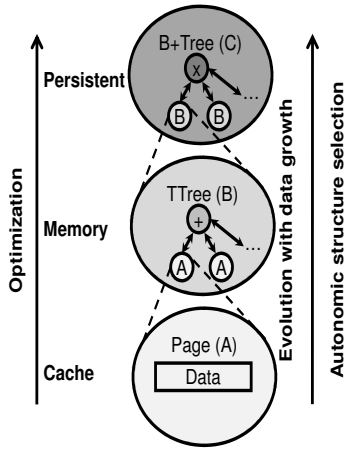

Figure 1: Evolving hierarchically-organized storage structure.

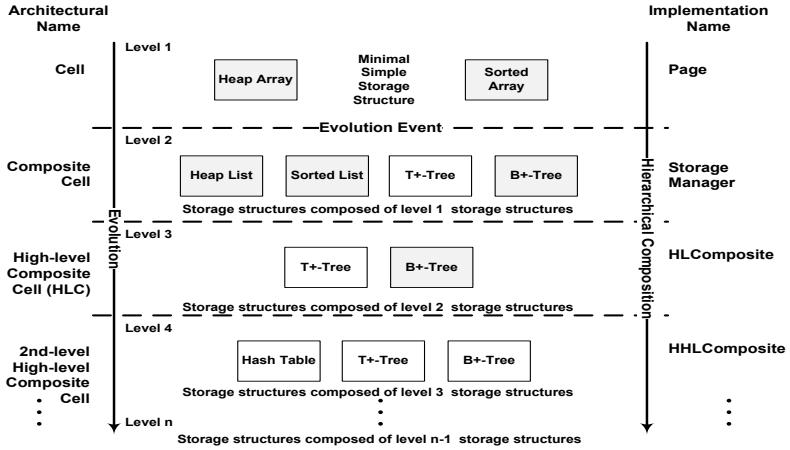

Figure 2: Evolving hierarchically-organized storage structure configurations.

be performed to evolve a storage structure. Heredity information means gathered statistics about a storage structure, e.g., workload type, data access pattern, previous evolution details, etc.

3.1 Mechanisms to alter an evolution path

An evolution path can be analyzed over multiple repetitions to identify either it is optimal or not. We call the repetition of an evolution path as evolution cycles and statistics generated over multiple evolution cycles are stored as part of heredity information. Evolution cycles occur frequently when evolving storage structures are used for intermediate result materialization during query processing, but it might never occur for data storage scenario where data only increases. If an initial evolution path is found to be sub-optimal, i.e., either for the query-intensive data the query time is high or for the write-intensive data, the write and update time is high, we propose three mechanisms to alter an evolution path, i.e., Disaster, War, and Preaching.

Disaster : Disaster mechanism identifies a new evolution path based on the heredity information and mutation rules. Once started, it executes running DBMS operations and queues the new arriving requests. It instantiates the new instance of storage structure based on a new evolution path and transfers the data to new storage structure all-together. Old storage structure is considered dead after data transfer and is eliminated. Once data transfer is completed, queued requests of new DBMS operations are completed with/over the new storage structure. Disaster mechanism is optimal for storage structures with the small data and light workload. It consumes resources all-together to optimize the storage structure. Furthermore, it improves the performance for all workloads simultaneously as complete storage structure is evolved all-together. We term the time to adapt the new storage structure based on the updated evolution path as the "Revolution Period".

War : War mechanism works like disaster mechanism to identify a new evolution path. It has a different data transfer strategy. It transfers data recursively starting from fine-grained atomic cell-level[2](atomic storage structure unit, e.g., Page. See Figure 2) based on a new evolution path. One by one for each cell, data is transferred to a new cell, and then the old cell is considered dead and is eliminated. In war mechanism only the DBMS operations of a single cell are completed, and new operations are queued before the data transfer, which ensures that DBMS operations that do not involve the particular cell can be completed without any delay. We suggest that war mechanism is appropriate for medium-sized data storage with the medium-level workload.

Preaching : Preaching mechanism works like war mechanism except the difference that decision to transfer data to a new cell is decided according to the cell state. Preaching mechanism may take longer to change cells, but it attempts to ensure that minimum overhead is incurred for ongoing DBMS operations. A cell waits for data transfer until/unless either it is in waiting or idle state, or it contains the workload well within the threshold defined by the DBMS administrator.

4. EVALUATION

In this section, we present our evaluation of four important factors, i.e.,

- Impact of unused functionalities on storage structures performance.
- Impact of data size growth on storage structures performance.
- Impact of storage structure complexity on resource consumption.
- Performance improvement for evolving storage structures.

To present the impact of unused functionalities on storage structures performance, we used Berkeley DB as our data management solution. As shown in Figure 4, RECNO, Queue, Hash, and B+-Tree represent the storage structures of Berkeley DB that we used for evaluation. To analyze the impact of data growth on storage structures performance, the impact of storage structures complexity on resource consumption, and the performance gains of evolving storage structures, we used the implementation of storage structures in our Cellular DBMS prototype[3] [14].

[2]Cell is a Cellular DBMS architecture term used for an atomic, autonomic, and customized instance of an embedded database. For details please refer [14]

[3] "Cellular DBMS", `http://wwwiti.cs.uni-magdeburg.de/~srahman/CellularDBMS/index.php` (Please refer to weblink for all related publications and evaluation prototype

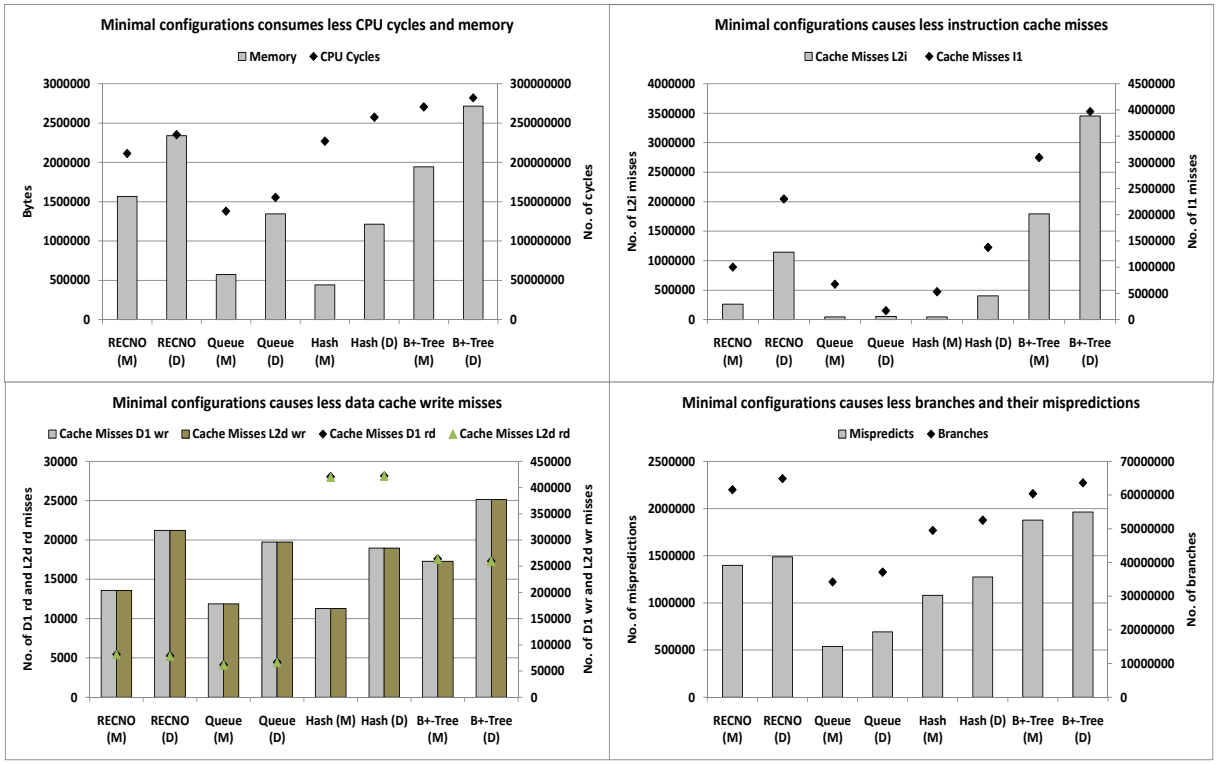

Figure 4: Micro benchmark results presenting the impact of unused functionalities on storage structures performance using the BerkeleyDB.

To compare the performance of different storage structures, we used CPU cycles, Heap, and Cache as resources. The reason for selecting these parameters is the change in bottlenecks. In the last two decades, processor speed has been increasing at a much faster rate of around 60% per year in comparison with the memory speed that increases around 10% per year [11]. Furthermore, with the increase in cache size, the latency for cache access is also increasing [6]. Now it is essential to make optimal use of increased processing power available to DBMS. Optimal processing power usage requires optimal cache usage by reducing the cache misses. It also requires efficient utilization of main memories.

For storage structures evaluation, we set up a micro benchmark with repeated insertion, selection, and deletion of key/-value pairs of data using API-based access method. The reason for using a custom micro benchmark is that we only want to focus on storage structures, and we are using storage structures with API-based access method(no query processing overhead). The data contain keys in ascending, descending, and random order, which also represents their insertion, selection, and deletion order in the database. All storage structures used in a micro benchmark operate in main-memory as we focused on the impact of storage structure complexity on processor, memory, and cache usage. Largest data storage unit contains 55.57 KB of data spreading over 4048 key/value pairs.

We used OpenSuse 11.2 operating on Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz with 4 GB of RAM. It contains two 32 KBytes 8-way set associative L1 instruction and data cache with 64-byte line size; and one 4 MB 16-way set associative L2 cache with 64-byte line size. We used Valgrind [15] to generate cache references and misses, and heap usage. We measured execution speed by taking the average of CPU cycles observed over multiple iteration of micro benchmark. All parameters presented in Figure 4, 6, and 7 are valid for comparison of structures and should not be considered as the benchmark for the Cellular DBMS performance. For better visibility of charts, we used few abbreviations that mean as follows: I=Instruction cache reference, D rd= Data (read) cache reference, D wr=Data (write) cache reference, I1= L1 Instruction cache miss, D1 rd=L1 Data (read) cache miss, D1 wr=L1 Data (write) cache miss, L2i=L2 Instruction cache miss, L2d rd=L2 Data (read) cache miss, and L2d wr=L2 Data (write) cache miss. For Berkeley DB, we used version 4.8.26.

In Figure 6 and 7, storage structures are similar as we defined in Table 1 except of two storage structures, i.e., HLC SL and HLC B+-Tree. HLC stands for High-Level Composite in the Cellular DBMS architecture. SL is short for Sorted List. HLC SL means a B+-Tree-based structure where each leaf node is a Sorted List, where as HLC B+-Tree means a B+-Tree-based structure where each leaf node is a B+-Tree. HLC SL and HLC B+-Tree storage structures are depicted in Figure 5.

To evaluate the impact of unused functionalities on storage structures performance, we tested all four Berkeley DB storage structures with two Berkeley DB configurations, i.e., Default(D) and Minimal(M). Default configuration contains all features of Berkeley DB by default, whereas for minimal configuration, we removed all removable features that include following flags: –disable-largefile, –disable-cryptography, –disable-hash, –disable-queue, –disable-replication, –disable-statistics, –disable-verify, –disable-partition, –disable-compression,

binaries.)

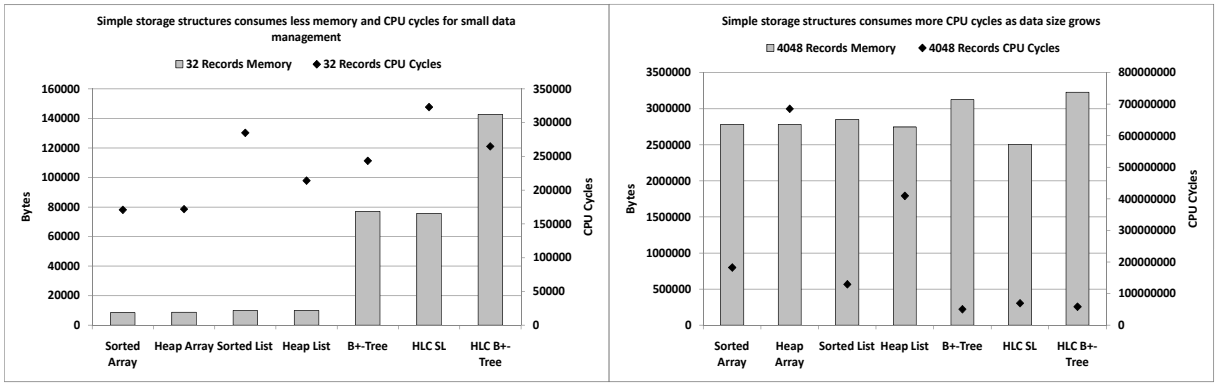

Figure 6: Micro benchmark results presenting the impact of data growth and increase in complexity on storage structures performance using the Cellular DBMS prototype.

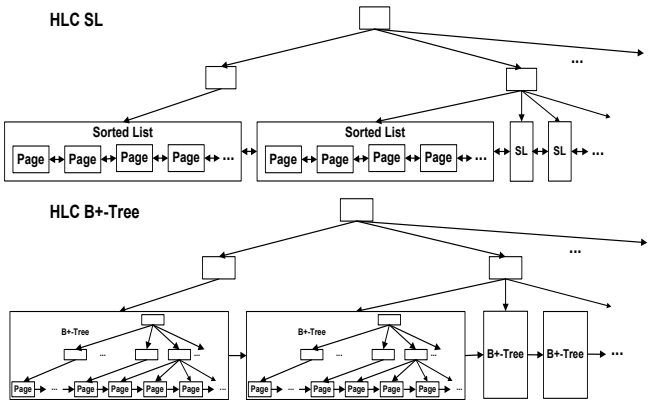

Figure 5: HLC SL and HLC B+-Tree storage structures in the Cellular DBMS prototype.

–disable-mutexsupport, and –disable-atomicsupport. For RECNO and Queue, –disable-queue flag is not used, whereas for Hash, –disable-hash is not used. It can be observed that minimal configuration consumes much fewer resources than the default configuration for all storage structures and thus have better performance. Furthermore, it can also be observed that for our benchmark data of 4048 records, simple storage structures perform better than B+-Tree. To further strengthen this argument, we also tested the Cellular DBMS prototype storage structures with different data sizes of 32 and 4048 records. It can be observed from Figure 6 that simple and small storage structures, such as Sorted Array and Heap Array perform better than other complex storage structures by consuming less memory and CPU cycles for small data of 32 records. For 4048 records, simple storage structures underperform, whereas other storage structures perform better. We also tested these structures on data size of 100000 records. Hash table performed best for 100000 records data size. Then we also tested these structures on data size of 10000000 records. B+-Tree outperformed hash table for 10000000 records data size (Test binaries and results for 100000 and 10000000 records are available at the Cellular DBMS weblink).

To evaluate the performance gain using evolving storage structures, we tested our benchmark using evolving versions of B+-Tree, HLC SL, and HLC B+-Tree storage structures. It can be observed from Figure 7 that each evolving storage structure version performs better than normal structures in resource consumption and thus exhibit enhanced performance.

5. RELATED WORK

Hierarchically-organized storage structures have been in use in the data warehousing domain. Morzy et al. in [8] proposed hierarchical bitmap index for indexing set-valued attributes. Later, Chmiel et al. in [5] extended that concept to present hierarchically-organized bitmap index for indexing dimensional data. Bender et al. proposed the cache oblivious B-Trees [1] that performs the optimal search across different hierarchical memories with varying memory levels, cache size, and cache line size. Fractal prefetching B+-Trees [4] proposed by Chen et al. is the most relevant work for the evolving storage structures and is similar in concept to cache oblivious B-Trees with an additional concept of prefetching. Fractal prefetching B+-Trees are optimized for both cache and disk performance, which is also a goal for the evolving storage structures.

Evolving storage structures use hierarchical organization differently. It allows the use of different storage structures at the different level of hierarchy and increases the hierarchy with data growth autonomically making efficient use of underlying hardware. Evolving storage structures expose the consistent interface for different storage structures across the hierarchy to hide the internal complexity of storage structures from other functionalities and end-user. It also allows the use of any storage structure atomically, i.e., we can also make use of single page as a storage structure for managing small database. Evolving storage structures evolve from simple to complex storage structures with data growth to ensure optimal usage of resources.

6. CONCLUSION AND FUTURE WORK

We presented Evolving Storage Structures as a storage structure for self-tuning data management systems. We evaluated different storage structures using micro benchmark to raise the awareness of impact of unused functionalities, storage structure complexity, and data size growth on storage structures resource consumption and performance. Our results also showed reduced resource consumption and improved performance for evolving storage structures. Evaluation of evolving storage structures using TPC Benchmark^{TM}H (TPC-H) [13] is in progress.

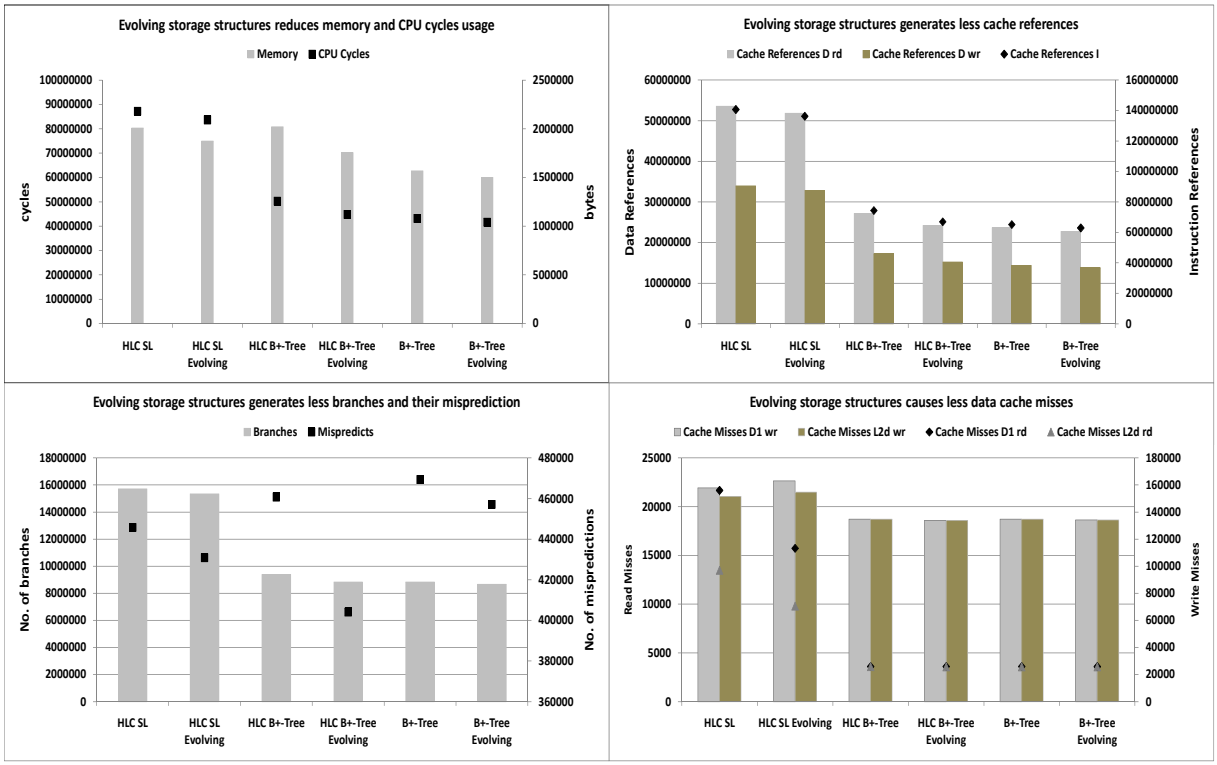

Figure 7: Micro benchmark results presenting performance improvement and reduced resource consumption using evolving storage structures.

7. ACKNOWLEDGMENTS

Syed Saif ur Rahman is a HEC-DAAD Scholar funded by Higher Education Commission of Pakistan and NESCOM, Pakistan.

8. REFERENCES

[1] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 399–. IEEE Computer Society, 2000.

[2] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. Biennial Conf. on Innovative Data Systems Research*, pages 225–237, January 2005.

[3] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 1–10. Morgan Kaufmann Publishers Inc., 2000.

[4] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-Trees: optimizing both cache and disk performance. In *Proc. Int'l Conf. on Management of data*, pages 157–168. ACM, 2002.

[5] J. Chmiel, T. Morzy, and R. Wrembel. HOBI: Hierarchically Organized Bitmap Index for Indexing Dimensional Data. In *Proc. Int'l Conf. on Data Warehousing and Knowledge Discovery*, pages 87–98. Springer-Verlag, 2009.

[6] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proc. Biennial Conf. on Innovative Data Systems Research*, pages 79–87, January 2007.

[7] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 294–303. Morgan Kaufmann Publishers Inc., 1986.

[8] M. Morzy, T. Morzy, A. Nanopoulos, and Y. Manolopoulos. Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes. In *ADBIS*, volume 2798, pages 236–252. Springer-Verlag, 2003.

[9] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. Annual Conf. on USENIX Annual Technical Conference*, pages 43–43. USENIX Association, 1999.

[10] Oracle Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[11] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17:34–44, March 1997.

[12] G. Saake, M. Rosenmüller, N. Siegmund, C. Kästner, and T. Leich. Downsizing Data Management for Embedded Systems. *Egyptian Computer Science Journal (ECS)*, 31(1):1–13, January 2009.

[13] TPC-H. http://www.tpc.org/tpch/.

[14] S. S. ur Rahman, V. Köppen, and G. Saake. Cellular DBMS: An Attempt Towards Biologically-Inspired Data Management. *Journal of Digital Information Management*, 8(2):117–128, April 2010.

[15] Valgrind. http://www.valgrind.org.