*International Journal of Computer Applications (0975 – 8887)*
*Volume 127 – No.2, October 2015*

# Map vs. Unordered Map: An Analysis on Large Datasets

Akanksha Bindal
Department of Computer Engineering
Delhi Technological University
New Delhi, India

Prateek Narang
Department of Computer Engineering
Delhi Technological University
New Delhi, India

S. Indu, PhD
Department of Electronics and Communication
Delhi Technological University
New Delhi, India

## ABSTRACT
In the current scenario, with the transpiring big data explosion, data sets are often too large to fit completely inside the computers´ internal memory. In efficient processes, speed is not an option, it is a must. Hence every alternative is explored to further enhance performance, by expanding in-place memory storage that enables more data to be resident in the memory, eliminating operation latency, and even deploying an in-memory database (IMDB) system where all the data can be kept in memory. However, the technique of in-memory data handling is still at an infant stage and not viable in the current scenario. To tackle this problem a hierarchical hashing scheme is discussed where only one component of a big data structure resides in the memory. In this paper two data structures are explored: 1) Map which is implemented as self-balancing binary search trees or more commonly Red Black Trees and 2) Unordered Map which is based on hashing with chaining technique. Serialization and deserialization operations are also performed to free the internal memory and preserve the data structure object for later use. Operations such as read, write are performed, along with documentation of the results and illustrations of visual representations of the two algorithmic data structures.

## General Terms
Advanced Data Structures, Algorithms, Memory management et al.

## Keywords
Map, Unordered Map, Boost C++, Serialization, Hierarchical hashing, Memory management

## 1. INTRODUCTION
In most large scale applications across multiple domains, data sets are too large to fit completely inside the computer's internal memory. A major performance barrier is observed due to the I/O communication constraints between fast internal memory and slower external memory (such as secondary storage devices). In this paper, the state-of-the art in the design and performance of STL data structures for handling large data sets is analyzed.

### 1.1 Horizons: Big Data Explosion
Big Data Explosion has stimulated much research into constructing systems that support the handling of low latency in internal-external memory communication processes and improve real time data analytics. As a result of the high latency to disk access, existing hard disk based systems fail to provide timely response. The unacceptable performance was initially encountered by Internet companies such as Amazon, Google, Facebook and Twitter, [2] but is now also becoming a hindrance for other organizations which desire to provide a reliable real-time service (e.g., real-time auction services, advertising, social gaming). For example, trading companies must detect a sudden change in the stock market prices and react immediately (in milliseconds), which seems unlikely to achieve using traditional disk-based processing systems [3]. In order that we meet the strict requirements for analyzing large amounts of unstructured data in real-time and cater to requests in milliseconds, an IMDB system that loads the entire data into the Random Access Memory (RAM) as and when required, is necessary. Jim Grays farsighted insight that Memory is the new disk, disk is the new tape is slowly seen materializing into reality today [4]. We are living in an era that will gradually lead to the replacement of disks by improved memory constructs resulting in the role of disks becoming more archival. In recent decades, new breakthroughs are being created due to the emergence of multi-core processors and the availability of large amounts of main memory at tumbling costs. For instance, memory storage capacity and bandwidth have been doubling roughly every three years, while its price has been dropping by a factor of 10 every five years [3]. Evolution of database systems over the last few decades is primarily driven by significant progress in hardware, availability of a large amount of data, emerging applications, collection of data at an unprecedented rate and so on. The landscape of the two types of database systems is increasingly divided based on the assortment of available application (i.e., applications depending on structured data, graphical data, and stream data input). Figure 1 shows state-of-the-art commercial and related systems for disk-based memory and in-memory database systems and operations. Applications today range from large scale computation intensive activities such as Big Data problems solved by using technologies like Map Reduce in Hadoop to fast real-time based applications that require lossless data transmit at the risk of low computational power. With such a diverse range of applications, memory requirements are scaling at an unprecedented rate. In efficient day-to-day processes, speed is not an option, it is a must. Hence every alternative is explored to further enhance performance, by expanding in-place memory storage that enables more data to be resident in the memory, eliminating operation latency, and even deploying an in-memory database (IMDB) system where all the data can be kept in memory. However, the technique of in-memory data handling is still at an infant stage and not viable in the current scenario. To tackle this problem a hierarchical hashing scheme is discussed where, at a time, only one component of a big data structure resides in the memory.

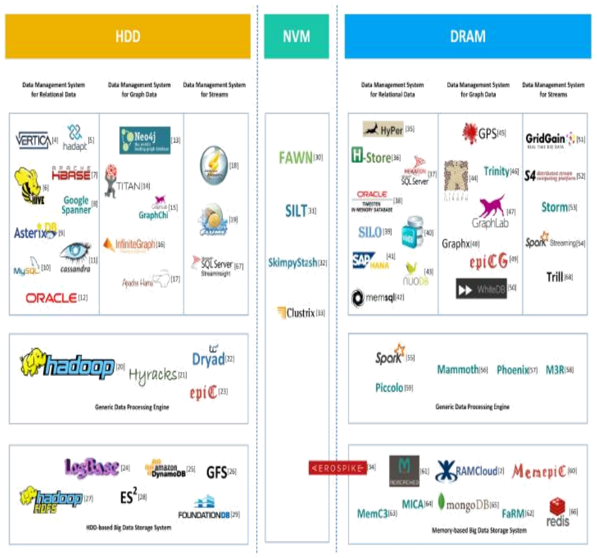

HDD
NVM
DRAM
Data Management System for Relational Data
Data Management System for Graph Data
Data Management System for Streams
Generic Data Processing Engine
HDD-based Big Data Storage System
Memory-based Big Data Storage System

**Fig 1. The (Partial) Landscape of Disk-Based Management Systems**

## 2. STANDARD TEMPLATE LIBRARY CONTAINERS

## 2.1 STL Container: Unordered Map

Unordered maps fall under the subset of associative containers that use a pair of a key and a mapped value to store the corresponding elements. In an unordered map, the key value is usually used to uniquely identify the element, while the mapped value stores the content associated to this key. These data structures allow for fast retrieval of individual contained elements based on their mapped keys. Internally, the elements in the unordered map are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average)[8]. Results indicate that Unordered map constructs are much faster than map containers to access individual elements by their key value, but are generally less efficient for range traversal through a subset of their elements. However when sorted data is required in the output, Unordered Maps even after applying additional sorting algorithms, are equally efficient as Red Black Trees(that automatically produce sorted results).

### *2.1.1 Hashing using Unordered Map:*

In computing applications, a hash table (hash map) is a data structure used to implement a structure that can map keys to values, broadly known as the associative array. A hash table relies on a proper hash function to calculate an index into a set of buckets or slots, from which the correct value can be then retrieved. In ideal conditions, no hash collision (different keys that are assigned by the hash function to the same bucket) occurs and the hash function assigns each key to a single bucket. But it is possible that the hash function generates identical hash values for two keys causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions will occur and must be accommodated in some way. In this paper, it is assumed to have access to hash functions that behave as truly random functions, independent of the sequence of operations to be performed. This means that any hash function value h(x) is uniformly random across the input domain x and independent of hash function values on elements different from x. The efficient functioning of a hash table depends on the fact that the table size is proportional to the number of entries. When the size of hash table is fixed, and common structures are used, hash table works similar to a linear search, except with a better constant factor. Further for achieving better memory management implicit resizing is carried out in the hash table. Resizing is accompanied by a full or additive table rehash operation whereby existing items are mapped to new bucket locations. To constrain the amount of memory wasted due to empty buckets, some implementations also shrink the size of the table, followed by a rehash, when items are deleted. From the point of space-time trade-offs, this operation is similar to the de-allocation in dynamic arrays.

## 2.2 STL Containers: Map

Maps are another subset of associative containers that adhere to a specific order while storing elements formed by a key and a mapped value pair. In a map, the key value are generally used to sort the objects and uniquely identify the elements, while the content associated to this key is shown by the mapped values. The types of key and mapped value may differ. Internally, the elements in a map are always sorted by its key following a specific criterion involving strict weak ordering indicated by its internal comparison object [10]. Map containers are generally slower than unordered map containers to access individual elements by their key. However, they allow range iteration on subsets based on the underlying order, which is missing from unordered map containers. Maps are typically implemented as self balancing binary search trees.

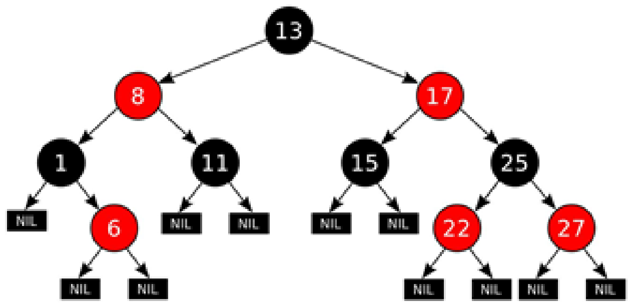


**Fig. 2. An example of a red-black tree**

### *2.2.1 Using Red Black Trees:*

A red black tree is a type of binary search tree with an extra bit of data per node, its color, which can be either red or black. This extra bit of information ensures an approximately balanced tree by restricting how nodes on any path from the root to the leaf are colored. Thus, it is a data structure which works as a self-balancing binary search tree. Balance is preserved by coloring each node of the tree with one of two colors (standardized as ’red’ and ’black’) in a way that satisfies certain properties, which effectively constrain how unbalanced the tree can become in the worst case. Whenever the tree is modified, the coloring properties of the new tree are stored by subsequent rearrangement and repainting of nodes. The properties are designed in such a way that this rearrangement can be performed efficiently. The balancing of the tree though not perfect, is good enough to guarantee searching in *O(log n)* time, where n is the total number of elements in the tree. Tree insertion and deletion operations, along with the node rearrangement and recoloring, are also performed in *O(log n)* time.

| | **Map** | **Unordered_map** |
|---|---|---|
| **element ordering** | strict weak | n/a |
| common implementation | balanced tree or red-black tree | hash table |
| search time | **log(n)** | O(1) if no hash collisions, Up to O(n) if there are hash collisions, O(n) when hash is same for any key |
| **Insertion time** | log(n) + rebalance | Same as search |
| **Deletion time** | log(n) + rebalance | Same as search |
| needs comparators | only operator < | only operator == |
| **needs hash function** | **no** | yes |
| common use case | when good hash is not possible or too slow. Or when order is required | In most other cases |
| | | |

**Fig. 3. Comparison of Map and Unordered Map STL Containers**

## 3. EXPERIMENT

### 3.1 Data Set

Data sets containing 1 billion entries are considered. Each entry has a unique integer as its key and an 8 bit character as value. This data is generated using a Python script which selects few million random entries for a given range (For example: 2-6 Million). Test is performed on ranges of size 1 million, 2 million, 4 million and 8 million entries. For example, with respect to 8 million range ranges of 0-8 million, 16-24 million and so on are considered.

### 3.2 Methodology

In this data set hashing of up to 16 million entries is performed on both Unordered Maps and Maps. Performance of these data structures degrade as the number of entries increase. Hence, a hierarchical hashing scheme based on range of integer keys is used. The first hash function is based on integer key range which decides the hash table a particular entry will go into. Instead of populating a single data structure object with huge data (around 2 GB), multiple hash table objects (for both unordered map and map) are used for storing the data. The hashing at the second level occurs in STL Maps and Unordered Maps which use the inbuilt hashing functions. Due to large data and limited memory (RAM), all objects cannot reside in memory at the same time. Hence, serialization of objects which are not required at that time is carried out. *Def:* The process of transforming an object in memory into a stream of bytes in order to store the object or transmit it over a database, network is called serialization. The primary purpose is to store the state of the object to be able to recreate it when needed. The reverse operation is called deserialization. This helps in freeing the internal memory (RAM) and preserving the data in secondary storage for later use. Boost Library in C++ is used for the purpose of serialization and deserialization.

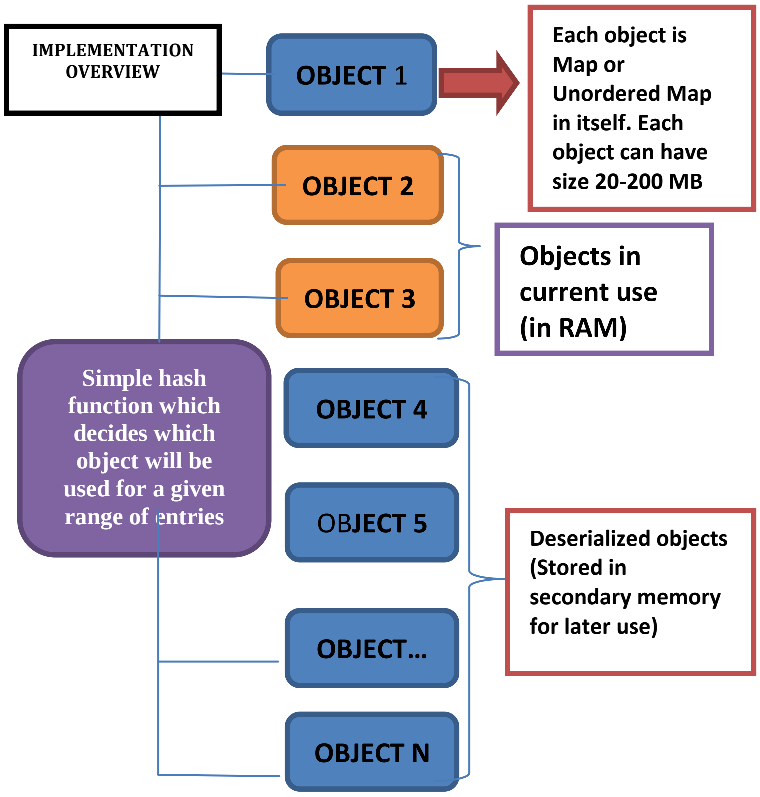

IMPLEMENTATION OVERVIEW
OBJECT 1
Each object is Map or Unordered Map in itself. Each object can have size 20-200 MB
OBJECT 2
OBJECT 3
Objects in current use (in RAM)
Simple hash function which decides which object will be used for a given range of entries
OBJECT 4
OBJECT 5
Deserialized objects (Stored in secondary memory for later use)
OBJECT...
OBJECT N

**Fig. 4. Overview of internal hierarchical hashing scheme**

### 3.3 Experimental Results

Write operations are performed on each data structure object (implemented using both Map and Unordered Map) which can handle data entries having a total size ranging from 20MB to 200 MB. Data stored in even 10 such objects takes 2 GB of space which is not available in RAM of normal computers. Hence serialization operations are performed after writing each object and the corresponding CPU clock time is noted. Later, reading data from these hash maps is done by loading each object in memory through deserialization. Reading operation is performed in which each read takes O(1) time. The read can be either sequential or in any arbitrary order. The unordered map produces output in a random order which needs to be sorted based on keys. Thus some additional sorting time is required for it but still it is more efficient than maps. Four graphs are plotted for a single data structure object which can handle data ranging from 20 - 200 MB. Multiple such objects were used to perform the entire test but with a single object residing in memory at a given time. The graphs display the average time taken for various operations by one such object. Results are displayed in Fig *5*, *6*, *7* and *8* for Update, Serialization, Loading and Deserialization operations respectively. The values for the dataset ranging from 1 million to 16 million are tabulated and displayed in *Table 1*.

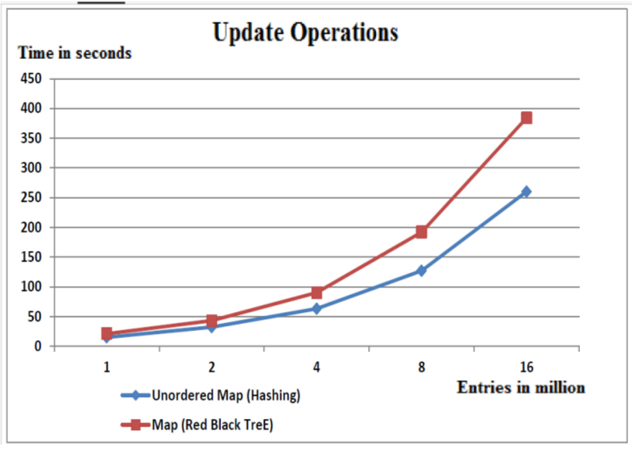


**FIG. 5. Comparison based on updating the data structure object. Unordered Maps perform faster and Maps become slow as number of entries increases**

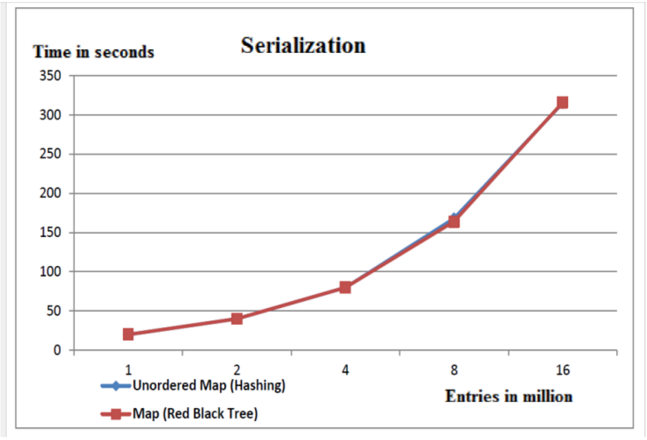


**FIG. 6. Comparison based on serialization of data structure object. Both Map and Unordered Map take the same time**

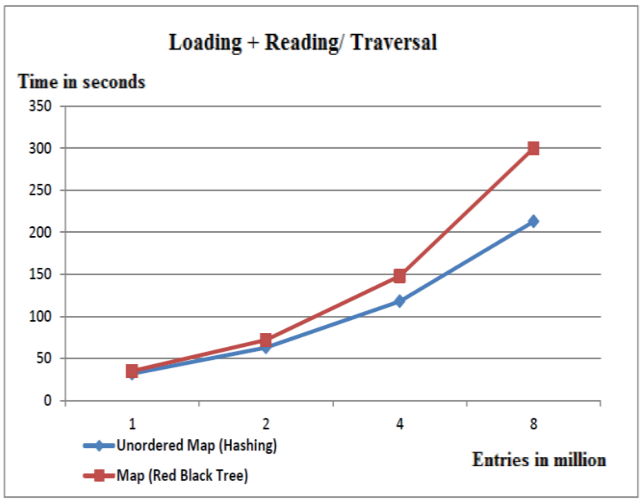
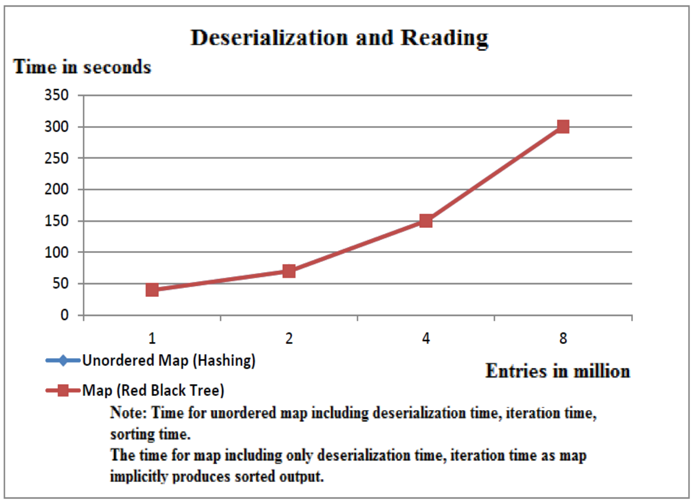


**FIG. 7. Comparison based on deserializing and iterating over the entire object (Excluding the sorting time for Unordered Maps) This shows that Unordered Map is faster to traverse than a Map**



Note: Time for unordered map including deserialization time, iteration time, sorting time.
The time for map including only deserialization time, iteration time as map implicitly produces sorted output.

**FIG. 8. Comparison based on deserializing and iterating over the entire data object (Including the sorting time for Unordered Maps) Note: Time for Unordered Map including deserialization time, iteration time and sorting time. The time for Map including only deserialization time and iteration time as map implicitly produces sorted output.**

**TABLE 1: Table displaying the time to perform update and read operations for entries from 1 million - 8 million**

| Update Operations | | |
|---|---|---|
| Entries in Million | Time in seconds | |
| | Unordered Map | Map |
| 1 | 15 | 21 |
| 2 | 32 | 43 |
| 4 | 63 | 90 |
| 8 | 127 | 192 |
| 16 | 260 | 384 |

| Update + Serialization | | |
|---|---|---|
| Entries in Million | Time in seconds | |
| | Unordered Map | Map |
| 1 | 35 | 41 |
| 2 | 72 | 83 |
| 4 | 146 | 170 |
| 8 | 295 | 352 |
| 16 | 575 | 700 |

| Serialization + Deletion from memory | | |
|---|---|---|
| Entries in Million | Time in seconds | |
| | Unordered Map | Map |
| 1 | 20 | 20 |
| 2 | 40 | 40 |
| 4 | 80 | 80 |
| 8 | 168 | 164 |
| 16 | 315 | 164 |

| Deserialization + Reading/Iteration | | |
|---|---|---|
| Entries in Million | Time in seconds | |
| | Unordered Map | Map |
| 1 | 32 | 35 |
| 2 | 63 | 72 |
| 4 | 118 | 148 |
| 8 | 213 | 300 |

## 4. CONCLUSION

Results indicate that Unordered Maps perform better than Maps in most cases for operations like updating, reading, erasing and complete iteration over the data structure, as illustrated in Table 1. Maps become slow as the internal tree size increases. This happens because tree balancing operations are more frequent when the entries are sequential. As the number of entries increase the tree size also increase and thus operations becomes slower. However, this is not the case for Unordered Maps. The time to find the right bucket and push the data into it is almost constant (having a O(1) amortized time complexity) regardless of the number of entries to a particular limit. After a particular limit is reached, rehashing is done automatically/implicitly. Rehashing operations take time but frequency of rehashing is very small as compared to balancing operations in tree based maps. So, unordered maps perform much better. It is also found that both data structures take the same time for Serialization and Deserialization operations. In case sorted output (based on keys) from Unordered Maps is desired, sorting algorithms are implemented to sort the random data obtained from them. In this case, they take the same time as reading from Maps, which implicitly produce sorted output.

## 5. ACKNOWLEDGMENTS

Our sincere thanks to all the professors in the Programming Lab at College for their timely support and inputs.

## 6. REFERENCES

[1] Jeffrey Scott Vitter, Purdue University ”Dealing with massive data” http://www.cs.purdue.edu/homes/jsv/ Papers/Vit.IO survey.pdf 2006.

[2] G. DeCandia, D. Hastorun, M. Jampani et al., Dynamo: Amazons highly available key-value store, OSR, 2007.

[3] Hao Zhang, Gang Chen,Kian-Lee Tan et al., ”In-Memory Big Data Management and Processing: A Survey”

[4] S. Robbins, Ram is the new disk, *InfoQ News*, Jun. 2008.

[5] J. Ousterhout, P. Agrawal, D. Erickson et al., The case for ramclouds: Scalable high-performance storage entirely in dram, *OSR,* 2010.

[6] F. Li, B. C. Ooi, M. T. O zsu, and S. Wu, Distributed data manage-ment using mapreduce,*ACM Comput. Surv.*, 2014.

[7] Wikipedia-Wikipedia.org.

[8] Standard C++ Library reference: http://www.cplusplus.com/reference/unorderedmap/unor dered map/ Cpp Reference Documentation

[9] The Boost C++ Libraries: http://theboostcpplibraries.com/ Boost C++ Documentation

[10] Standard C++ Library reference: http://www.cplusplus.com/reference/map/map/ Cpp Reference Documentation

[11] HP, Vertica systems, 2011. [Online]. Available: http://www.vertica. com

[12] D. J. DeWitt, R. H. Katz, F. Olken et al., Implementation techniques for main memory database systems, *in SIGMOD 84*, 1984.

[13] R. B. Hagmann, A crash recovery scheme for a memory-resident database system,*TC,* 1986.

[14] T. J. Lehman and M. J. Carey, A recovery algorithm for a high performance memory-resident database system, in *SIGMOD 87*, 1987.

[15] M. H. Eich, Mars: The design of a main memory database machine, in Database Machines and Knowledge Base Machines.*Springer US,* 1988.

[16] D. J. DeWitt and J. Gray, Parallel database systems: The future of high performance database systems, *CACM*, 1992.

[17] S. Wu, B. C. Ooi, and K.-L. Tan, Online aggregation, in Advanced Query Processing. Springer Berlin Heidelberg, 2013

[18] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan, Distributed online aggregations, in PVLDB 09, 2009.

[19] T. J. Lehman and M. J. Carey, A study of index structures for main memory database management systems, in PVLDB 86, 1986.

[20] J. Rao and K. A. Ross, Cache conscious indexing for decision-support in main memory, in PVLDB 99, 1999.

[21] D. B. Lomet, S. Sengupta, and J. J. Levandoski, The bw-tree: A b-tree for new hardware platforms, in ICDE 13, 2013.

[22] T. Brown, F. Ellen, and E. Ruppert, A general technique for non-blocking trees, in PPoPP, pp. 329342, ACM, 2014.

[23] T. Brown, Personal homepage.

[24] Oracle, Concurrentskiplistmap, 2014.

[25] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, Early experience with a commercial hardware transactional memory im-plementation, tech. rep., Sun Microsystems, Inc., 2009..

[26] A. C. Yao. On random 2-3 trees. Acta Informatica, 9, 159170, 1978.

[27] K.-Y. Whang and R. Krishnamurthy. Multilevel grid filesA dy-namic hierarchical multidimensional file structure. In Proceedings of the International Symposium on Database Systems for Advanced Applications, 449459. World Scientific Press, 1992.

[28] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, chapter 9, 431524