Towards Efficient Main-Memory Use For Optimum Tree Index Update

Laurynas Biveinis
lauras@cs.aau.dk
Supervised by Simonas Šaltenis
simas@cs.aau.dk
Computer Science Department
Aalborg University

ABSTRACT

An emerging class of database applications is characterized by frequent updates of low-dimensional data, e.g. coming from sensors that sample continuous real world phenomena. Traditional persistency requirements can be weakened in this setting of frequent updates, emphasizing a role of the main-memory in external storage index structures and enabling a higher update throughput. Moreover, in order for an index to be suitable for practical applications, efficient past-state queries should be supported without significantly penalizing other operations.

These issues are not adequately addressed in the database research. We report on the R^R-tree—our first step towards resolving them. Based on this, we outline a number of concrete short-term and more abstract longer-term future research directions.

1. INTRODUCTION

A rapid development of sensor and communication technologies makes it increasingly feasible to capture and query large amounts of data that capture continuously-changing real world properties.

Location-Based Services (LBS) is a representative example of such update-intensive applications, where positions of a large number of GPS-equipped moving objects are tracked on a central server. Position updates in such scenarios are issued very frequently to maintain location data within acceptable accuracy bounds. For example, vehicles travelling in a semi-urban environment have to issue an update every fifteen seconds on average to maintain a tracking accuracy of 200 meters [25].

Traditionally the research of the spatial indexing has considered mostly static data and consequently focused on an efficient support of queries. This is exemplified by properties of one of the most predominant spatial indexes, the R-tree [4, 11], which supports efficient queries, however the update performance has been recognized as inadequate [16].

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

Furthermore, the amount of the available main-memory in practical settings is continuously increasing, however this fact is largely ignored in the research of disk-based index structures. An efficient index structure should be able to use any allocated amount of the main-memory to maximize performance [1]. However, as demonstrated later, the existing approaches that do use the main-memory suffer from the drawbacks of requiring some minimum amount of the main-memory to work, or the opposite, not being able to use all the available main-memory. Moreover, those approaches do not constitute a systematic study. Finally, it should be noted that a generic way to utilize the main-memory—the LRU cache—is not effective when the goal is to speed up updates [6].

The role of the main-memory is further emphasized by an opportunity to relax persistency requirements in scenarios with high update ratios. Even in an event of a system crash, current positions of indexed objects will be received after a relatively short time interval. Thus in a setting of relaxed persistency, each update does not need to be logged to the disk immediately on arrival and can be handled purely in the main-memory at that time. This can greatly increase the update throughput of an index. However, there is little existing research discussing such setting, with exceptions being [10, 28].

Finally, indexing only the current state of moving object positions is rarely a whole solution. More often, an efficient support of past-state queries that does not penalize current-state queries and updates too much is equally important. The research in this area has started to appear only very recently [19] and the state of the art is not satisfactory.

To summarize we make the following observations:

1. Processing high rates of updates on index structures is essential to support new applications that rely on tracking of continuously changing real-world phenomena.

2. Persistency requirements in frequent update scenarios can often be relaxed to assume relaxed persistency only.

3. The main-memory is increasingly available in large quantities and taking into account the full memory hierarchy becomes essential.

4. Past-state queries are potentially as much important as current-state queries.

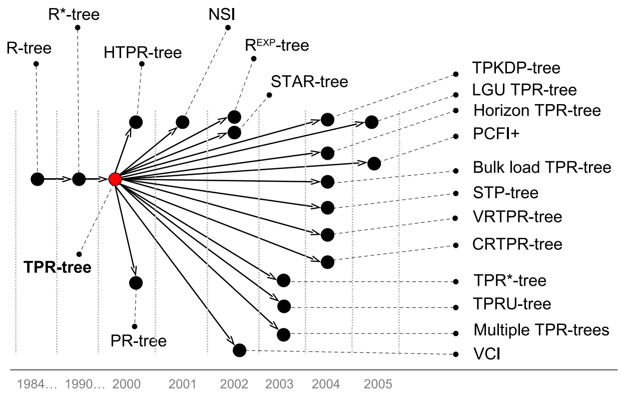


Figure 1: TPR-tree successors

We argue that to date these issues have been mostly ignored or addressed inadequately in the database research.

The first three of the above observations were started to be addressed in the R^R-tree [6]. This work demonstrates promising preliminary results as well as raises a number of immediate research problems to explore. In the following, we outline these problems as well as longer-term research issues.

The rest of this paper is organized as follows. Section 2 broadly overviews the related work in the proposed research areas. Section 3 outlines main contributions of the R^R-tree that provide background for future research. Then Section 4 discusses future research directions. Finally, Section 5 concludes.

2. RELATED WORK

Techniques for the bulk-loading of data structures are relevant for the problem of efficient updates. Notably, Choubey et al. [7] have proposed algorithms that aim to perform many insertions on an existing index structure in one go. However, these algorithms are not directly applicable in our setting because they focus on the search performance of the resulting structure, whereas the performance of the bulk-loading itself is of a secondary importance. In addition, they do not consider deletions.

An approach to dealing with frequent updates of continuous variables such as positions is to model the positions as linear functions of time instead of standard constant functions of time. When representing a position of an object by a linear function as, e.g., proposed by Wolfson et al. [26], the modeled position stays close to the actual for longer periods of time. It has been shown that, for a range of reasonable accuracy thresholds, the number of updates needed to maintain positions of vehicles is reduced by almost a factor of three [25]. The use of such linear functions on the top of the R-tree was first proposed by Šaltenis et al. [22] and subsequently explored by others (Figure 1).

The problem of frequent updates in the R-tree was recently tackled by Lee et al. [14], who present the most competitive proposal for bottom-up updates of R-trees. Such approach avoids potentially expensive top-down traversals and exploits the update locality of continuous variables. However, an auxiliary data structure, required in order to access the bottom level, uses a significant amount of space and is disk based. Because of this, updates still take at least 3 I/Os. Additionally, the proposal includes a main-memory data structure that compactly stores a summary of the tree. This structure helps save I/O in case of non-local updates, but uses a fixed amount of the main-memory.

Arge has proposed a general tree buffering technique that associates a buffer with every non-leaf node [2]. Operations are then not performed immediately, but are placed in the buffers. Once a buffer is full, its contents are moved to the buffers at the next level in its subtree. The main-memory is used for the buffer emptying. A drawback of this approach is that it cannot answer queries immediately. Arge et al. have also applied the technique to R-trees [3], and Van den Bercken and Seeger [8] have explored similar techniques.

Lin and Su propose the LGU algorithm that combines buffer-tree and bottom-up techniques [16]. They attach disk-based insertion buffers to the non-leaf nodes and perform insertions top-down in batches. To avoid multiple partial traversals for a single deletion, deletions are gathered in a main-memory deletion buffer that is applied lazily and directly to the leaf-level nodes. A main-memory based leaf-level access table facilitates this. A major drawback of this approach is that the size of the main-memory based leaf-level access table is the same order of magnitude as the size of indexed data. For example, for R-trees, the size of this access table is one third of the size of the index.

A recent approach by Xiong and Aref [27] significantly lowers an average update cost by performing deletions in the main-memory. However, insertions are performed using an ordinary R-tree algorithm, so there are no further performance gains for these. A main-memory data structure, called update memo, is responsible for keeping information about deletions as well as about the latest and obsolete data entries.

The AGILE method [9] addresses an issue of balancing conflicting performance requirements for update and query operations in a more general setting of information filtering. This approach proposes augmenting existing data structures in a way that selectively increases the update performance at a cost of losing the query accuracy. This proposal mentions implementing AGILE on the top of the R-tree as a part of their future work. The outline of such implementation suggests having buffers on internal nodes that store frequently updated objects. A detailed comparison with our approach is impossible at this time as AGILE has not actually been implemented on top of the R-tree yet.

Most of these approaches assume strong persistency requirements. A notable exception is LUGrid [28], which adopts an approach that is closely related to the R^R-tree, assuming a grid as an underlying data structure. While conceptually simple, this approach cannot be directly compared to R-tree-based indexes, as the R-tree and the grid have different application areas with the most notable difference being that grids are not suited for indexing objects with extents.

In a context of B-trees, the main-memory utilization for an LRU buffer has been recognized as an inadequate solution and LRU-based smart buffering schemes have been proposed instead [20, 21]. However, these do not offer order of magnitude improvements over a simple LRU scheme.

In a survey, Graefe [10] presents a general overview of techniques for speeding up B-tree updates. A number of presented techniques, such as buffering of insertions, buffering in separate structures and batching of updates are directly relevant to our setting. Moreover, a concept of *non-logged B-trees* discusses a possibility of relaxing persistency requirements that is very similar to our setting. Finally, an idea of *differential indexes* is presented by Graefe, where all incoming updates are gathered in a separate data structure. Other results in this area include [18, 23]. However, to the best of our knowledge, no existing work considers a possibility of *partial* emptying of the main-memory index.

An efficient main-memory use has been most explored in the context of main-memory databases, where all data resides in the main-memory. While in traditional DBMSs, main-memory page buffers are organized as simple collections of pages, main-memory databases employ more elaborate index structures to optimize the main-memory access and the CPU performance [15, 24].

To be efficient, such structures are usually made *cache-conscious*, i.e., they take into account parameters of upper levels of the memory hierarchy. In the same way as data is transferred in blocks between the disk and the main-memory, data is also transferred in blocks of cache-line size between the main-memory and the CPU cache. Alternatively, main-memory index structures can be made *cache-oblivious*, so that they perform well in various different memory hierarchies without taking a cache line size into account as an explicit parameter.

Most of the previous research on the main-memory indexing assumes that a main cost of index operations is due to CPU cache misses and fails to account for a CPU computation cost of the indexing operations. Only recently [12] has it been recognized that the CPU cost is as important as memory hierarchy properties in reasoning about main-memory index performance. This issue becomes even more important with indexes that perform complex calculations in their operations, such as the TPR-tree [22].

The research is lacking in the area of an efficient past-state query support that still enables efficient current-state queries and updates. To date, the only significant proposal which addresses these problems is the R^{PPF}-tree [19]. However, it does not address the issue of efficient main-memory use.

3. THE R^{R}-TREE

Below we will present general principles of the R^{R}-tree with a small example and selected performance results. For a complete treatment, the reader is referred to [6].

The R^{R}-tree builds on the principles of the R-tree [4, 11]. It is a height balanced tree with indexed data stored in leaf nodes. Non-leaf nodes contain entries pointing to the next level nodes and associated Minimum Bounding Rectangles (MBRs) that spatially contain data in the subtree below.

The R^{R}-tree consists of two main structures: a main-memory R-tree, termed *operation buffer* and a disk-based R-tree. The disk R-tree is a standard R-tree. In contrast, the operation buffer contains operations that have not been yet performed on the disk R-tree. It collects all incoming update operations, both insertions and deletions, thus its leaf node entries are augmented with an additional flag to differentiate them apart. In contrast to some of the related work [2, 3, 14, 16, 27], there is no lower or upper bound for an amount of the main-memory that should be allocated to the operation buffer.

In addition, some incoming operations are allowed to fully complete in the main-memory without involving the disk tree at all. If an incoming update finds an existing "opposite" operation in the buffer that concerns the same object, both are simply deleted in a so-called *annihilation.* If the size of the operation buffer is being increased, more and more operations are completed this way, up to a point where all data fits in the main-memory. Then the R^{R}-tree operates purely as a main-memory R-tree. At the other extreme, if no main-memory is available, the R^{R}-tree behaves mostly as a regular disk R-tree.

To answer queries, both the disk-based tree and the operation buffer are queried. This is one of reasons why the operation buffer is organized as a main-memory index. This approach is related to Arge's Buffer Tree [2], with a fundamental difference that the Buffer Tree uses disk-based buffers, thus working in a full-persistency setting, at a cost of performance, as well as not supporting online queries.

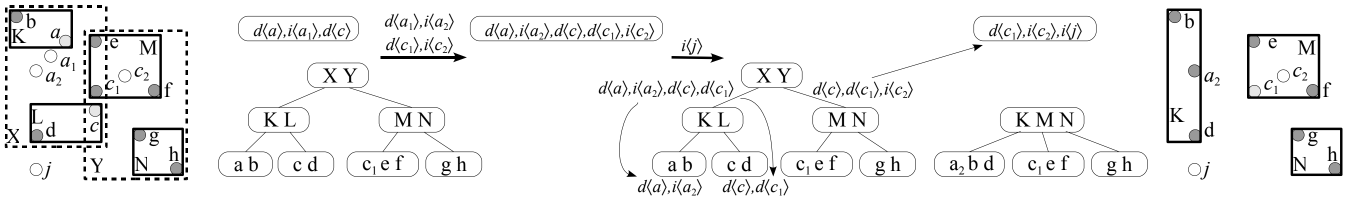
When the operation buffer fills all the available main-memory, some or all of its operations have to be executed on the disk tree in a so-called *buffer emptying.* This is done by clustering its operations into groups and performing the largest groups (or, as a generalization, groups larger than some threshold value k) in bulk on the disk. Disk I/O costs are significantly reduced compared to an approach of executing updates one-by-one, because if k operations from the buffer need to access the same disk tree node, disk accesses are shared between them: only one I/O is performed instead of k. On the other hand, when operations are performed in bulk, the disk tree update algorithms are more complicated than the R-tree ones, because the number of various cases that must be handled increases.

3.1 Example

Figure 2 shows a small example where a number of update operations is performed, causing a buffer emptying. The leftmost part of the figure shows positions of eight objects stored in an R-tree. The four empty circles (a_1, a_2, c_2, j) represent insertions that are in the buffer or the future updates that will be discussed in the following. The light gray circles (a, c) represent positions that are in the disk tree, but that have corresponding deletion entries in the buffer. In the example, the object a moves from a to a_1 to a_2, and the object c moves from c to c_1 to c_2. We assume a maximum fan-out of 3 and a minimum fan-out of 2 for the trees. The buffer has room for 5 entries and the operation threshold is 4.

Note that the initial state of the tree holds two positions, c and c_1, for the object c. This happened after a part of the buffer containing an insertion of c_1 was emptied, while a deletion of c was left in the buffer. Next, a and c update their positions, resulting in the following sequence of updates: $del\langle a_1\rangle$, $ins\langle a_2\rangle$, $del\langle c_1\rangle$, $ins\langle c_2\rangle$. The deletion of a_1 and a corresponding insertion already in the buffer annihilate each other. The remaining three operations are inserted into the buffer.

An insertion of a new object j triggers a buffer emptying. Operations are divided into two groups: one for the node X and one for the node Y. Note that, due to the overlap between MBRs of X and Y, $del\langle c\rangle$ and $del\langle c_1\rangle$ are copied into both groups. Because the group for the node Y has fewer than k elements, its entries are put back into the buffer. The operations in the other group proceed down the tree to the nodes K and L, as shown in the figure. Note that in the node L, $del\langle c_1\rangle$ is unsuccessful, but $del\langle c\rangle$ is successful and



$d\langle a\rangle, i\langle a_1\rangle, d\langle c\rangle$
$d\langle a_1\rangle, i\langle a_2\rangle$
$d\langle c_1\rangle, i\langle c_2\rangle$
$d\langle a\rangle, i\langle a_2\rangle, d\langle c\rangle, d\langle c_1\rangle, i\langle c_2\rangle$
$i\langle j\rangle$
$d\langle c_1\rangle, i\langle c_2\rangle, i\langle j\rangle$
X Y
K L
M N
K M N

Figure 2: Performing a set of operations on the R^R-tree

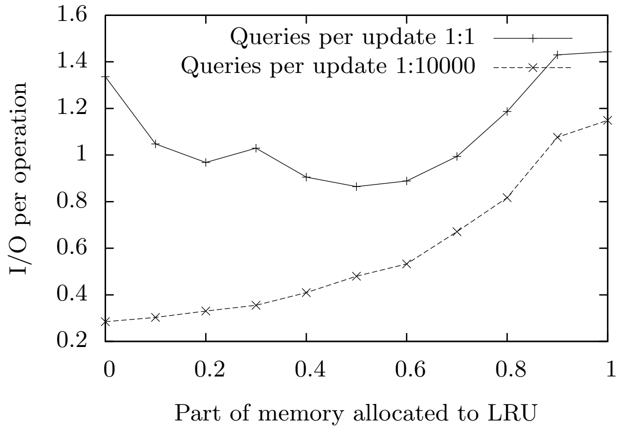


Figure 3: I/O performance relation to varying main-memory divisions

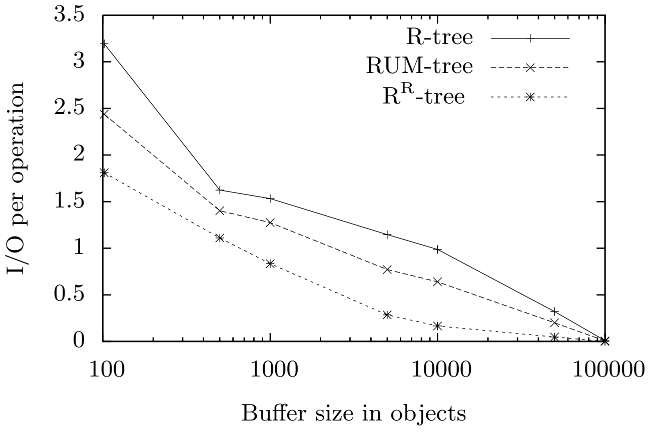


Figure 4: Scalability with respect to available main-memory

its copy is removed from the buffer.

The performed operations result in a number of structural changes to the disk tree. Node L is rendered underfull and is merged with the node K. As a result, the node X contains only one child; thus, X is replaced by its child, reducing the height of the subtree, which is then inserted into the node Y, to one. Finally, the single-entry root is removed, finishing the buffer emptying. At last, the operation that caused the buffer emptying, $ins\langle j\rangle$, is inserted into the buffer. The resulting disk tree and the buffer are shown at the rightmost end of the figure.

Note that the demonstrated shrinking (and growing) of subtrees is very rare for realistic node sizes and workloads that mostly contain deletion-insertion pairs.

3.2 Experimental Results

To demonstrate the viability of the R^R-tree approach, selected experimental results from [6] are presented. First, Figure 3 confirms that the operation buffering is a more effective way to use the main-memory than a write-back LRU cache in a setting with frequent updates. Furthermore, even in a setting with an equal number of updates and queries, a combination of operation buffer and an LRU cache is the most effective way of using the main-memory.

Figure 4 compares the R^R-tree with a plain write-back LRU-cached R-tree and the RUM-tree [27] in settings with varying amounts of the main-memory. Since the RUM-tree itself uses a fixed amount of the main-memory, it has also been extended with an LRU buffer to be able to utilize a larger amount of the main-memory. Based on the results, we conclude that the R^R-tree is more effective at utilizing any amount of the main-memory available. The rightmost point on the graph corresponds to a situation where all data fit in the main-memory. Because of an operation buffer or a write-back LRU cache, all three trees operate as main-memory trees in this setting.

4. RESEARCH DIRECTIONS

In this section we will discuss future research directions. We start with concrete issues directly arising from the R^R-tree research in Section 4.1.1. Then we discuss a generalization of operation buffering techniques to a broader class of tree indexes in Section 4.1.2. Section 4.2 takes a deeper look at main-memory issues and finally Section 4.3 addresses an efficient current and past state indexing.

4.1 Efficient Usage of the Main-Memory for Disk-Based Indexes

4.1.1 R^R-Tree Research Challenges

There is a number of immediate challenges that have to be addressed for the R^R-tree to fully unleash its potential. First, the cost model in [6] has to be improved to become simpler, solvable analytically and more precise. The "back of the envelope" main-memory buffer dynamics calculations in [6] can be replaced with a much more precise Markov chain based model (*random walk*).

Further, currently proposed techniques to set operation group size thresholds are not flexible enough. Intuitively the thresholds should be larger at upper tree levels (i.e. equal to the size of the largest group in the buffer) and smaller at the lower tree levels (i.e. it is almost universally unprofitable to execute groups containing only one operation). Thresholds below the root level are further complicated by a total cost of returning operations back to the buffer. While simple experimental trials show that dynamic thresholds are indeed profitable, the question of cost-modelling them and coming up with universal threshold determination heuristics remains open.

An interesting alternative that avoids the threshold issue altogether is to replace the single main-memory tree with individual variable-size buffers for all intermediate nodes of the disk tree. It is expected that this scenario would simplify the cost model greatly, at an expense of complicating R^{R}-tree algorithms.

4.1.2 *Generalization*

Although the operation buffering techniques of the R^{R}-tree work with R-trees, they do not depend on any core R-tree property. Thus, generalizing the techniques to a *grow and post* class of trees [17] is natural.

In particular, we plan to explore the B-tree in the context of the GiST [13] framework. We plan to develop a B-tree-index-based indexing technique that relies on a main-memory B-tree to buffer incoming operations and partial buffer emptying to selectively execute them on the disk B-tree. If the R^{R}-tree results [6] are indicative, we expect this technique to outperform related work by one or two orders of magnitude in terms of disk I/O. Notable research questions here include modelling of a buffer state and development of an optimal analytically-grounded buffer emptying technique.

While the results of this research would be applicable in a much broader area than spatial indexing, it would still further the main goal of improving the spatial continuously-changing data indexing by enabling an use of B-tree-based spatial indexes, e.g. with space filling curve indexing techniques.

4.2 Main-Memory Indexing of Continuously Changing Data

An operation buffer structure in the main-memory should have good main-memory index properties, especially with large amounts of the main-memory. Techniques of making index structures cache-conscious will be investigated. Moreover, since cache line sizes are usually substantially smaller than disk block sizes, main-memory index trees tend to have nodes of a much smaller fan-out than disk-based ones. That, in turn, results in taller trees, meaning that heuristic-based insertion algorithms of multi-dimensional index structures designed for index trees of just a few levels may not work well. Thus the heuristics will have to be adjusted to perform well in such cases. Moreover, as an alternative to cache-conscious data structures, cache-oblivious ones could be explored.

Finally, we plan to evaluate a CPU cost of complex indexing techniques, in particular the ones employing linear approximations [22]. Such techniques enable maintaining accuracy requirements with fewer updates at a cost of being CPU-intensive. This trade-off will have to be evaluated and perhaps will lead to different design decisions when developing such structures. In particular, we plan to look at how to simplify the heuristics-based index algorithms to make them less CPU-intensive, perhaps by using a simpler structure as a base index, for example, the grid. Another research topic of interest is an investigation of bottom-up techniques in this setting.

4.3 Efficient Current-State Indexing with an Historical Queries Support

Finally we plan to extend indexing techniques with an efficient past-state query support that still enables efficient updates and current-state queries and effectively uses the available main-memory. An "obvious" solution of having two separate indexes—one for past and one for current state—suffers from significant drawbacks of a data communication overhead between the indexes and possible object trajectory discontinuities.

The only significant proposal which addresses these problems is the R^{PPF}-tree [19]. However, the proposed solution is complex and also does not address the challenge of efficient usage of the available main-memory.

Some principles of *cache-oblivious* data structures (e.g. [5]) are indirectly relevant to this research. Of particular interest is the transformation of tree nodes between different memory hierarchy levels that have different native node (disk page, cache line, etc.) sizes. This can potentially be applied to transform and move tree nodes between the trees that are at different memory hierarchy levels (for example, from the main-memory to the disk) and thus have different node sizes.

To conclude, the goal of this research direction is to develop an efficient spatiotemporal indexing solution that is able to use the main-memory effectively (as a special case, it could be an efficient main-memory-only structure) and is able to handle a high rate of incoming updates, while having an adequate query performance, both of the current and the past state.

5. CONCLUSION

Inspired by a development of database applications, such as LBS, which monitor large amounts of continuously changing data, we present the need to efficiently support a large number of updates in index structures. Moreover, we observe that traditional persistency requirements can be weakened if relaxed persistency is enough, enabling a significantly higher update throughput. Furthermore, the main-memory is playing an increasingly important role in disk-based indexing, and effects of the whole memory hierarchy must be considered. Finally, the current state indexing is inadequate and a robust index should be able to answer both past and current state queries efficiently, while still supporting efficient updates.

We maintain that these issues have not been adequately addressed in literature and we have started addressing them in the R^{R}-tree [6], built on the main ideas of operation buffering and the group update.

Based on promising preliminary results from the R^{R}-tree research, the following future research directions are proposed. First, to address issues immediately raised by the R^{R}-tree, we propose to develop a cost model and heuristics to direct dynamic operation group size thresholds. The next step is to generalize the proposed techniques for the family of grow and post trees, particularly for the B-tree. Moreover, main-memory index structures should be investigated to consider not only the memory hierarchy, but the CPU cost as well, especially if linear predictions or other

CPU-intensive algorithms are used. Finally, we propose to develop an index that satisfies a need for applications to have an efficient past state query support, while still supporting efficient current state queries and updates.

6. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? *VLDB J.*, pp. 266–277, 1999.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms (extended abstract). In *Proc. WADS*, Volume 955 of *Lecture Notes in Computer Science*, pp. 334–345. Springer Verlag, 1995.

[3] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD*, 19(2):322–331, 1990.

[5] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. IEEE FOCS* p.399. Washington DC, USA, 2000.

[6] L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-memory operation buffering for efficient R-tree update. In *Proc. VLDB '07*, 2007.

[7] R. Choubey, L. Chen, and E. A. Rundensteiner. GBI: A generalized R-tree bulk-insertion strategy. In *Proc. SSD '99*, pp. 91–108. Springer, 1999.

[8] J. Van den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *Proc. VLDB '01*, pp. 461–470, 2001.

[9] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: adaptive indexing for context-aware information filters. In *Proc SIGMOD '05*, 2005.

[10] G. Graefe. B-tree indexes for high update rates. *SIGMOD Rec.*, 35(1):39–44, 2006.

[11] A. Guttman. *R*-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, 14(2):47–57, 1984.

[12] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious B+-trees. *SIGMETRICS Perform. Eval. Rev.*, 31(1):283–294, 2003.

[13] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. VLDB '95*, pp. 562–573, 1995.

[14] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: A bottom-up approach. In *Proc. VLDB '03*, pp. 608–619, 2003.

[15] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proc. VLDB '86*, pp. 294–303, 1986.

[16] B. Lin and J. Su. Handling frequent updates of moving objects. In *Proc. ACM CIKM*, pp. 493–500. 2005.

[17] D. B. Lomet. Grow and post index trees: Roles, techniques and future potential. In *Proc. SSD '91*, pp. 183–206. 1991.

[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[19] M. Pelanis, S. Šaltenis, and C. S. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM Trans. Database Syst.*, 31(1):255–298, 2006.

[20] G. M. Sacco. Index access with a finite buffer. In *Proc. VLDB '87*, pp. 301–309, 1987.

[21] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. Database Syst.*, 11(4):473–498, 1986.

[22] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc7 SIGMOD '00*, pp. 331–342, 2000.

[23] D. G. Severance and G. M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.

[24] In-memory databases: The catalyst behind real-time trading systems. TimesTen white paper, http://www.timesten.com, 2005.

[25] A. Čivilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE TKDE*, 17(5): 698–712, 2005.

[26] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proc. IEEE SSDBM*, pp. 111–122, 1998.

[27] X. Xiong and W. G. Aref. R-trees with update memos. In *Proc . ICDE '06*, p. 22, 2006.

[28] X. Xiong, M. F. Mokbel, and W. G. Aref. LUGrid: Update-tolerant grid-based indexing for moving objects. In *Proc. MDM*, p. 13, 2006.