# Efficient Evaluation of Hyper-Rectangular Blocks of Update Operations Applied to General Data Structures

Mugurel Ionut Andreica, Andrei Grigorean, Andrei Parvu, Nicolae Tapus

# Efficient Evaluation of Hyper-Rectangular Blocks of Update Operations Applied to General Data Structures

Mugurel Ionuţ Andreica*, Andrei Grigorean†, Andrei Pârvu* and Nicolae Ţăpuş*
*Computer Science Department
Politehnica University of Bucharest,
Splaiul Independenţei 313, sector 6, Bucharest, Romania, RO-060042
Email: mugurel.andreica@cs.pub.ro, andrei.prv@gmail.com, nicolae.tapus@cs.pub.ro
†Faculty of Mathematics and Computer Science,
University of Bucharest,
Str. Academiei 14, sector 1, Bucharest, Romania, RO-010014
Email: andrei.grigorean@gmail.com

***Abstract*—In this paper we present novel solutions for the following problem: We have a general data structure $DS$ and a set of update operations organized into a $D$-dimensional cube of side $N$ (thus, there are $N^D$ update operations). We are interested in efficiently evaluating range queries of the following type: compute the result of applying all the update operations within a hyper-rectangular block of the $D$-dimensional cube to $DS$ (considering that $DS$ is initially empty). The result of applying the updates consists of computing some aggregate values over the data structure. We consider that the order of applying the updates is irrelevant (i.e. the update operations are commutative) and that the aggregate results corresponding to a block of updates cannot easily be computed by combining the results of a set of sub-blocks whose disjoint union is $B$. However, the results can be efficiently maintained after each update operation, if the operations are performed sequentially in any order.**

***Keywords*—data structures, hyper-rectangular blocks of updates, sequence of updates, range query, block partitioning.**

## I. Introduction

Applying a large number of update operations to a data structure and afterwards computing an aggregate result is an important scenario which has not received sufficient attention in the scientific literature so far, particularly when the results of different sets of updates cannot be easily aggregated. In this paper we address the situation in which the update operations are placed in the cells of a $D$-dimensional cube of side length $N$ and we want to efficiently evaluate the result of the application of a subset of these update operations on an initially empty data structure. The subset of update operations consists of a hyper-rectangular block of the $D$-dimensional cube containing the update operations. We will consider both the online and the offline case and we will also place emphasis on scenarios where $D$ is small (e.g. when $D = 1$ the update operations are placed in a sequence and we are interested in applying contiguous subsequences of update operations to the data structure). Our results hold for any type of data structure and any type of updates specific to it. In this paper we will present solutions which are capable of evaluating the result of the applications of the update operations without actually applying each update operation on the data structure each time. In order to achieve this we first need to preprocess the $D$-dimensional cube of update operations and precompute multiple values.

The rest of this paper is structured as follows. In Section II we define the problem statement clearly. In Section III we present a solution for the online case (i.e. the queries are answered as they come, one at a time). In Section IV we improve the memory requirements of the solution presented in Section III for the case $D = 1$ and all the queries are available offline. In Section V we discuss several applications of our solutions. In Section VI we present experimental evaluation results for the solutions proposed in this paper. In Section VII we discuss related work and in Section VIII we conclude and discuss future work.

## II. Problem Statement

We consider that we have a data structure $DS$ on which we can apply certain update operations. The update operations are placed in the cells of a $D$-dimensional cube of side length $N$. We will denote by $Op(c(1), \ldots, c(D))$ the update operation located in the cell $(c(1), \ldots, c(D))$ of the cube ($1 \leq c(i) \leq N$, $1 \leq i \leq D$). The data structure is capable of efficiently maintaining some aggregate result values after applying each update operation. The result values are independent of the order in which a given subset of update operations are applied (i.e. the update operations are commutative).

We are interested in efficiently answering queries of the following type: Given a hyper-rectangle $\prod [l(i), h(i)]$ ($1 \leq l(i) \leq h(i) \leq N$, $1 \leq i \leq D$), apply all the update operations $Op(c(1), \ldots, c(D))$ with $l(i) \leq c(i) \leq h(i)$ ($1 \leq i \leq D$) to an initially empty data structure and return the result values maintained by the data structure after applying all the operations.

## III. Online Solution

We will consider that the side of the cube in each dimension is split into groups of size $K$ (except possibly for the last group, which may contain fewer than $K$ elements). We will define $G(i) = (i-1)/K + 1$ as the group to which the coordinate $i$ belongs. We will now consider the $D$-dimensional cube $CG$ of side length $(N + K - 1)/K$, where $CG(c(1), \ldots, c(D))$ is a sub-cube of the original cube consisting of the operations $Op(c'(1), \ldots, c'(D))$ such that

$G(c'(i)) = c(i)$ $(1 \leq i \leq D)$. Note that we consider integer division throughout this paper.

We will compute an instance of our data structure for each possible hyper-rectangle contained in $CG$. Let $DSI(a(1), \ldots, a(D), b(1), \ldots, b(D))$ be the data structure after applying all the update operations from the entries $(c(1), \ldots, c(D))$ of $CG$ such that $a(i) \leq c(i) \leq b(i)$ $(1 \leq i \leq D)$. For $a(i) = b(i)$ (for all $1 \leq i \leq D$) we will compute the data structure instance by sequentially applying all the update operations in the range. When $a(i) < b(i)$ for at least one value of $i$ $(1 \leq i \leq D)$ we can either compute the data structure instance from scratch (by applying all the update operations in the range) or we can create a copy of $DSI(a(1), \ldots, a(D), b(1), \ldots, b(i) - 1, \ldots, b(D))$ and then apply to this copy the remaining update operations (i.e. all the update operations corresponding to full-subcubes $CG(c(1), \ldots, c(D))$ where $c(i) = b(i)$ and $a(j) \leq c(j) \leq b(j)$ for $(1 \leq j \leq D,\ j \neq i)$. Note that we assumed that the data structure instances are computed as follows: we choose the tuple $(a(1), \ldots, a(D))$ and then the tuples $(b(1), \ldots, b(D))$ are considered in ascneding order of their sum (i.e. $b(1) + \ldots + b(D)$), breaking ties arbitrarily in case of equal sums.

By using this preprocessing approach we end up computing $O((N/K)^{2 \cdot D})$ data structure instances. Each data structure instance can be computed with $O((N/K)^{D-1} \cdot K^D)$ applications of update operations to a data structure instance. Thus, the overall number of applications of update operations of the preprocessing algorithm is $O((N/K)^{3 \cdot D-1} \cdot K^D)$. A good choice of the parameter $K$ is $\sqrt{N}$. In this case the number of applied update operations is $O(N^{2 \cdot D-0.5})$. For $D = 1$, the overall number of update operations applied to the data structure is of the order $O(N \cdot \sqrt{N})$.

In order to answer a query for a hyper-rectangle $\prod [l(i), h(i)]$ $(1 \leq i \leq D)$ we will proceed as follows. We will start from a copy of $DSI(G(l(i)) + 1 (1 \leq i \leq D), G(h(i)) - 1 (1 \leq i \leq D))$ (if we have $G(l(i)) + 1 > G(h(i)) - 1$ for some value of $i$ then we start with an empty data structure). Then we will apply to the initial data structure all the update operations $Op(c(1), \ldots, c(D))$ $(l(i) \leq c(i) \leq h(i),\ 1 \leq i \leq D)$ such that $G(c(i)) = G(l(i))$ or $G(c(i)) = G(h(i))$ for at least one value of $i$. An easy way to accomplish this is by choosing each dimension $i$ independently and ranging all the coordinates $c(j \neq i)$ $(1 \leq j \leq D)$ between their lower and upper bounds. If $G(l(i)) = G(h(i))$ the coordinate $c(i)$ will be varied from $l(i)$ to $h(i)$. Otherwise, we first vary it from $l(i)$ to the last coordinate in the same group as $l(i)$ (i.e. the coordinate $G(l(i)) \cdot K$) and then we vary it from the first coordinate in the group $G(h(i))$ (i.e. $(G(h(i)) - 1) \cdot K + 1$) to $h(i)$. However, when $D \geq 2$, we need to make sure to not apply the same update operation multiple times (if needed, we can maintain a hash table with the update operations which were already applied for this query).

There are $O(2 \cdot D \cdot K^D \cdot (N/K)^{D-1})$ update operations left to be applied to the starting data structure for each processed query. When $K = \sqrt{N}$ the number of applied update operations needed for answering a query is $O(D \cdot N^{D-0.5})$ (plus we need to consider the time to create a copy of a data structure instance, in case the data structure is large). Note that the naive solution needs to apply $O(N^D)$ update operations in order to answer a query. When $D = 1$ we need only $O(\sqrt{N})$ applied update operations in order to answer a query using our solution.

## IV. Offline Solution for $D = 1$

For the case $D = 1$ we will provide an offline solution which uses less memory than the solution presented in the previous section. If we implement that solution we end up storing $O(N)$ data structure instances. We will see how we can solve the problem by using only one data structure instance in the offline case. When $D = 1$ we will denote the query interval $[l(1), h(1)]$ by $[l, h]$.

First of all, each query for which $h - l + 1 \leq 2 \cdot K$ will be processed directly by applying all the update operations on an initially empty data structure. If creating an empty copy of the data structure each time is too expensive then we can always use the same empty data structure and, after applying the update operations for a query, we will undo them in order to get back to an empty data structure. We discuss the possibility of supporting undo operations later in this section.

In the other cases we will sort the queries in ascending order of the value $G(l)$ (and, in case of ties, in ascending order of $G(h)$). Let's assume that we are now processing all the queries with $G(l) = i$. We will sort all these queries in ascending order of their $G(h)$ value. We will start with an empty data structure instance. Then, for $j = i + 2$ up to $j = (N + K - 1)/K$:

1) We will apply all the update operations $Op(c)$ with $G(c) = j - 1$ to our data structure instance.
2) We will consider all the queries with $G(l) = i$ and $G(h) = j$ (note that these queries are positioned consecutively in the sorted order, right after all the queries with $G(l) = i$ and $G(h) = j-1$). For each such query we start from our data structure instance and we apply to it the update operations $Op(c)$ with $c \geq l$ such that $G(c) = G(l)$ and all the update operations $Op(c')$ with $c' \leq h$ and $G(c') = G(h)$.
3) We compute the answer to the query (now the data structure instance has all the operations from the interval $[l, h]$ applied to it).
4) We restore our data structure instance to the state before Step 2. If the update operations are invertible then we can achieve this by applying the inverse update operation for each update operation applied in Step 2. Otherwise, in order to maintain the efficiency of our approach, the data structure needs to efficiently support an *undo* operation. For instance, many common data structures which can be efficiently made persistent can support an *undo* operation by reverting to the version before applying the updates from Step 2. In case the data structure does not have an undo operation we can use the generic algoritm described in subsection IV-A.

Note that the total number of operations applied for answering all the queries is $O(N \cdot \sqrt{N})$ plus $O(\sqrt{N})$ per query (when $K = \sqrt{N}$), which is basically the same as in the online solution. However, the memory consumption is reduced (only one data structure instance is maintained at all times).

### A. Generic Undo Algorithm

We need to be ale to undo the $O(K)$ operations performed on our data structure instance during Step 2 of the presented algorithm (or when handling queries for which $h - l + 1 \leq 2 \cdot K$). We can achieve this as follows. We will maintain a stack $S$, which is initially empty. While applying the update operations, every time a field of the data structure is modified we push on the stack the name of the field (or a reference to it) together with its old value. Then, in order to undo all the update operations, we simply pop the top element from the stack as long as the stack is not empty and set the corresponding field of the data structure to its old value (also stored in the stack). Note that it is possible for the same data structure field to appear multiple times in the stack, but this poses no problem. By using this method the overall theoretical time complexity of our solution does not increase. Moreover, this method is applicable to any data structure, as long as we have access to its implementation.

## V. Applications

In this section we present several applications of our solutions from the previous two sections.

### A. Counting Connected Components for Contiguous Subsequences of Edges

Let's consider that we have an undirected (multi-)graph with $N$ vertices and $M$ edges. Its edges are placed in a sequence (e.g. they are time-stamped and ordered in increasing order of the time stamps): $e(1), \ldots, e(M)$. We want to efficiently evaluate the following types of queries: Given an interval $[i, j]$, compute the number of connected components of the graph if only the edges $e(i), \ldots, e(j)$ are considered.

If we choose the data structure to be the disjoint sets Union-Find data structure [9], then we can consider each edge to be a union operation. Moreover, the data structure is augmented in order to maintain the number of current disjoint sets (in an empty data structure this number is $N$). After each successful union (i.e. when two different sets are joined together) the number of sets decreases by 1 (note that the number of disjoint sets at any time is exactly the number of connected components of the graph).

Thus, we can immediately apply our results for $D = 1$ to this problem in order to obtain a solution with $O(M \cdot \sqrt{M})$ applications of the union operation during the preprocessing stage and $O(\sqrt{M}))$ applications of the union operation for answering each query. Note that the offline solution can also be used. The Union-Find data structure has been augmented in order to support *Undo* operations [4], [7]. However, even without this augmentation, we can still apply the generic technique mentioned in the previous section.

This problem was also considered in [5], where a more efficient, but more specific solution was proposed.

### B. Counting Black Cells in Multidmensional Cubes under Range Set Updates

Let's consider that we have an $F$-dimensional cube $MC$ of side length $G$. Each cell of the cube can be either white or black (initially all the cells are white). An update operation specifies a hyper-rectangle fully contained in the cube and sets all the cells within that hyper-rectangle to the color black. A query asks for the total number of black cells in $MC$. An efficient data structure for supporting such range update operations is a multidimensional region quadtree. Each quadtree node is associated to a hyper-cube fully contained in the original cube. The root node of the quadtree is associated to the whole cube $MC$. The leaves of the quadtree correspond to the unit cells of the cube. All the nodes except for the leaves have $2^F$ children - each child is associated with a hyper-cube with side length equal to half of the side length of the parent's hyper-cube. The union of the zones of the children is equal to the zone of the parent. For a more detailed description of multidimensional quad-trees see [14].

The quadtree can support the update operation as follows. Each quadtree node will maintain the total number of black cells inside its zone and whether the zone is fully black or not (moreover, it will also store the total number of cells in its zone). Thus, the number of black cells stored by the root will represent the number of black cells in the whole cube. An update operation proceeds as follows. Let's denote by $H$ the hyper-rectangle corresponding to the update operation. We start with the root as the current node. Let's assume now that the current node of the quadtree which is being visited is $v$. If $v$ is marked as being fully black then we simply do not process $v$ further, because its zone is already all black. If $v$'s zone is not all black, then if $H$ fully includes $v$'s zone then we mark $v$'s zone as being fully black and we set its number of black cells to the total number of black cells of the zone. If, however, $H$ does not fully include $v$'s zone, then we consider each child $w$ of $v$. If $H$ intersects $w$'s zone, then we apply this procedure recursively with $w$ as the current node. Before making a recursive call we decrease $v$'s number of black cells by $w$'s (current) number of black cells. After returning from the recursive call we add the (now updated) number of black cells from $w$'s zone to the number of black cells corresponding to $v$.

After each update the total number of black cells in the whole cube can be found at the root of the multidimensional quadtree. Note that this procedure also supports range queries (i.e. compute the number of black cells in a given hyper-rectangular block $H$). The query algorithm is similar to the update algorithm. We start at the root and whenever we reach a node $v$ which is all black we add to the answer the number of cells in the intersection between $H$ and $v$'s zone. If $v$ is not all black then we apply the recursive procedure for all the children of $v$ which intersect $H$ (if any).

The range update operations can be placed in a sequence (for $D = 1$) or in a multidimensional cube (for $D \geq 2$) and we want to evaluate the effect a hyper-rectangle of update operations has on an initially white multidimensional cube $MC$. We can immediately apply our solutions presented in sections III and IV for this problem.

Note that instead of a multidimensional quadtree we could have used a *region* kd-tree (also named *bintree* in the literature [14]).

### C. Counting Black Cells in Multidmensional Cubes under Range Flip Updates

In this subsection we consider the same problem as in the previous subsection, except that a range update operation "flips" the color of all the cells from a given hyper-rectangle included in $MC$ (i.e. if the cell was white it become black, and if it was black it becomes white). These operations are

also commutative, just like the range set operations from the previous subsection.

We will use the same multidimensional quadtree data structure. We will first describe how an update operation can be efficiently supported by the quadtree. For each quadtree node we will maintain the same values as before. However, instead of maintaining the information whether the node is all black, we will maintain a *flipped* bit indicating whether the node was flipped and this flip operation was not propagated to its descendants in the tree (initially no node is flipped). The update procedure is also a recursive procedure starting at the root. Let's assume that the current node is $v$. If $v$ is marked as being flipped and $v$ has any children, then we unmark $v$ from being flipped and we change the value of the *flipped* bit of each child $w$ of $v$ (from 0 to 1, or from 1 to 0). Moreover, we set the number of black cells of each child $w$ of $v$ to the total number of cells in $w$'s zone minus the current number of black cells in $w$'s zone. Then, if $v$'s zone is fully included in the update hyper-rectangle $H$ we change $v$'s *flipped* bit value (from 0 to 1, or from 1 to 0) and then we set its number of black cells to the total number of cells in $v$'s zone minus the current number of black cells in $v$'s zone. If $v$'s zone is not fully contained in $H$, then we will recursively apply the update procedure for each child $w$ of $v$ whose zone intersects $H$. Just like in the previous subsection, before making a recursive call we decrease $v$'s number of black cells by $w$'s (current) number of black cells. After returning from the recursive call we add the (now updated) number of black cells from $w$'s zone to the number of black cells corresponding to $v$.

After each update operation the root of the multidimensional quadtree contains the number of black cells in the multidimensional cube $MC$. Like in the previous case, this data structure can also support range queries (i.e. compute the number of black cells in a given hyper-rectangle $H$). The query procedure is similar to the update procedure. We start at the root of the quadtree. Let's assume that the current node of the quadtree is $v$. If $v$ is marked as being flipped, then we proceed like in the update procedure (we change $v$'s *flipped* bit and the *flipped* bits and the number of black cells of $v$'s children). Then, if $v$'s zone is fully contained in $H$ we add to the answer the number of black cells stored by $v$. Otherwise we apply the query procedure recursively for each child $w$ of $v$ whose zone intersects $H$.

The scenarios applicable to the problem from the previous subsection are also valid in this case. Also, just like in the previous subsection, we could use a *region* kd-tree instead of a multidimensional quadtree without making any changes to the update and query procedures.

Some instances of the problems described in subsections V-B and V-C may appear when a multidimensional binary data cube (e.g. a black and white image for $F = 2$) needs to be updated quickly and multiple times. Such scenarios occur, for instance, when car driving environments are modeled and tracked (e.g. the driving environment is modeled as a grid which is updated dynamically [10]).

## VI. Experimental Evaluation

We implemented both the naive and our optimized solution for the problem presented in subsection V-A. The code was written in C/C++ and compiled with the G++ compiler version 3.3.1. The tests were run on a machine running Windows 7 with an Intel Atom N450 1.66 GHz CPU and 1 GB RAM. We considered a graph with $N = 4096$ vertices and $M = 65536$ randomly generated edges. We generated $65536$ random queries, with the length of each query interval randomly chosen between $LMIN$ and $M$. We considered three values for $LMIN$: 1, 1000 and 30000, and we ran 5 different tests (with the graph and the set of queries regenerated each time). We used $K = \sqrt{M} = 256$.

For $LMIN = 1$ the naive solution ran in $361.92$ seconds and our optimized solution (the offline version described in section IV) ran in $8.50$ seconds (the reported time is the total time for all the 5 tests). For $LMIN = 1000$ the naive solution took $384.44$ seconds and our optimized solution took $8.60$ seconds, while for $LMIN = 30000$ the naive solution took $843.63$ seconds and our optimized solution took $7.50$ seconds. We also considered an extra optimization for the naive solution: stop performing unions for each query when the number of connected components became 1 (because the number of connected components cannot decrease below 1). The total running times were: $264.97$ seconds (for $LMIN = 1$), $258.13$ seconds (for $LMIN = 1000$) and $403.70$ seconds (for $LMIN = 30000$).

We can see that our optimized solution is more than $40$ times faster than the standard naive solution and more than 30 times faster than the optimized naive solution.

## VII. Related Work

Our solution makes use of a multidmensional block partitioning technique, which, in itself, is not a novel idea. Single- and multi-dimensional block partitioning techniques have been used before for speeding up range query processing [1], [2], [3], [12], [13]. References [2], [3] also specifically consider the possibility of efficiently handling range updates. However, it is assumed that the effect of the updates can be efficiently aggregated, unlike the scenario considered in this paper (when the effect of a set of update operations cannot be summarized and represented with a significantly lower complexity than listing the individual update operations). We would like to take this opportunity to correct a small error from [3]. The first instruction of the **STrangeQueryNodeIncl(node, a, b, dr)** function should be: **if** (*node.dim* $>$ *1*) **then return mop(STrangeQuery**(*node.*$T_{covering}$, $l_{node.dim-1}$, $h_{node.dim-1}$, *dr*), *a, b*). Moreover, the *dr* parameter should be added as the last argument of all the **STrange*** functions and passed along whenever these functions are called. Also, the **STpushUpdates** function doesn't need to be used (called) in the case of the multidimensional segment tree.

The block partitioning technique has also been employed when only range queries over a set of unmodifiable data are of interest. For instance, range mode queries solutions using this technique were presented in [6]. This method was also used as part of a solution for efficienly answering range minimum queries in multidimensional arrays [15].

Our offline solution for the 1D case assumes that the data structure on which the update operations are applied can efficiently support *undo* operations. Although we presented a general *undo* method in this paper, some data structures have their own more specific (and potentially more efficient) undo operations. One such example was already mentioned in

subsection V-A for the Union-Find disjoint set data structure, for which efficient solutions which can undo any number of union operations at a time exist [4]. In general, data structures which support persistence can easily support undo operations, by simply switching back to the version of the data structure before the operations were performed. General techniques for making data structures persistent have been proposed in the literature [11]. However, full data structure persistence is not required for undo operations and may introduce unnecessary overhead or complications. The concept of semi-persistence was introduced in [8], which is particularly suited to the possibility of undoing operations on a data structure.

## VIII. Conclusions

In this paper we presented new solutions to the problem of efficiently applying subsets of update operations to a data structure, when the update operations are placed in a multi-dimensional cube and the subset of applied update operations is a hyper-rectangle contained in this cube. Our solutions are general enough to be applied to any data structure and any type of update operations. They are efficient particularly in the case when the result of applying a subset of update operations cannot be summarized and represented with a significantly lower complexity than that of explicitly representing the update operations themselves. Experimental results confirmed the practical improvements suggested by the theoretical analysis.

## References


[1] M.I. Andreica and N. Ţăpuş, *Time Slot Groups - A Data Structure for QoS-Constrained Advance Bandwidth Reservation and Admission Control*, Proceedings of the $10^{th}$ IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), pp. 354-357, 2008.

[2] M.I. Andreica, *Optimal Scheduling of File Transfers with Divisible Sizes on Multiple Disjoint Paths*, Proceedings of the $7^{th}$ IEEE Romania International Conference "Communications", pp. 155-158, 2008.

[3] M.I. Andreica and N. Ţăpuş, *Efficient Data Structures for Online QoS-Constrained Data Transfer Scheduling*, Proceedings of the $7^{th}$ IEEE International Symposium on Parallel and Distributed Computing (ISPDC), pp. 285-292, 2008.

[4] A. Apostolico, G.F. Italiano, G. Gambosi, M. Talamo, *The Set Union Problem With Unlimited Backtracking*, Computer Science Technical Reports, Paper 774, Purdue University, 1989.

[5] M.J. Bannister, C. DuBois, D. Eppstein and P. Smyth, *Windows into Relational Events: Data Structures for Contiguous Subsequences of Edges*, Proceedings of the $24^{th}$ ACM Symposium on Discrete Algorithms, 2013.

[6] T.M. Chan, S. Durocher, K.G. Larsen, J. Morrison and B.T. Wilkinson, *Linear-Space Data Structures for Range Mode Query in Arrays*, Proceedings of the $29^{th}$ Symposium on Theoretical Aspects of Computer Science, pp. 1-12, 2012.

[7] S. Conchon and J.-C. Filliâtre, *A Persistent Union-Find Data Structure*, ACM SIGPLAN Workshop on ML, 2007.

[8] S. Conchon and J.-C. Filliâtre, *Semi-Persistent Data Structures*, Proceedings of the $17^{th}$ European Symposium on Programming, pp. 332-336, 2008.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to Algorithms*, $2^{nd}$ edition, MIT Press, 2001.

[10] R. Danescu, F. Oniga and S. Nedevschi, *Modeling and Tracking the Driving Environment with a Particle Based Occupancy Grid*, IEEE Transactions on Intelligent Transportation Systems, vol.12 (4), pp. 1331-1342, 2011.

[11] J.R. Driscoll, N. Sarnak, D.D. Sleator and R.E. Tarjan, *Making Data Structures Persistent*, Journal of Computer and System Sciences, vol. 38 (1), pp. 86-124, 1989.

[12] H.-G. Li, T.W. Ling, S.Y. Lee and Z.X. Loh, *Range Sum Queries in Dynamic OLAP Data Cubes*, Proceedings of the $3^{rd}$ International Symposium on Cooperative Database Systems for Advanced Applications, pp. 74-81, 2001.

[13] C. K Poon, *Dynamic Orthogonal Range Queries in OLAP*, Theoretical Computer Science, vol. 296 (3), 2003.

[14] H. Samet, *Multidimensional Spatial Data Structures*, Handbook of Data Structures and Applications, Chapter 16, CRC Press, 2005.

[15] H. Yuan and M.J. Atallah, *Data Structures for Range Minimum Queries in Multidimensional Arrays*, Proceedings of the $21st$ ACM-SIAM Symposium on Discrete Algorithms, pp. 150-160, 2010.