A Real-Time Spatial Index for In-Vehicle Units

Magnus Lie Hetland[1] and Ola Martin Lykkja[2]

[1]Norwegian University of Science and Technology, mlh@idi.ntnu.no
[2]Q-Free ASA, Trondheim, Norway, ola.lykkja@q-free.com

Abstract

We construct a spatial indexing solution for the highly constrained environment of an in-vehicle unit in a distributed vehicle tolling scheme based on satellite navigation (GNSS). We show that an immutable, purely functional implementation of a high-fanout quadtree is a simple, practical solution that satisfies all the requirements of such a system.

1 Introduction

Open road tolling is an increasingly common phenomenon, with transponder-based tolling proposed as early as in 1959 [1], and a wide variety of more complex technologies emerging over the recent decades [2]. One of the more novel developments is the use of satellite navigation (GNSS), with geographical points and zones determining the pricing [see, e.g., 3]. In this paper, we examine the feasibility of maintaining a geographical database in an in-vehicle unit, which can perform many of the tasks of such a location-based system independently.

This is a proof-of-concept application paper. Our main contributions can be summed up as follows: (*i*) We specify a set of requirements for a spatial database to be used in the highly constrained environment of a real-time, low-cost in-vehicle unit in Section 2; (*ii*) we construct a simple data structure that satisfies these requirements in Section 3; and (*iii*) we tentatively establish the feasibility of the solution through an experimental evaluation in Section 4. In the interest of brevity, some technical details have been omitted. See the report of Lykkja [4] for more information.

2 The Problem: Highly Constrained Spatial Indexing

The basic functionality of our system is to retrieve relevant geographic (geometric) objects, that is, the tolling zones and virtual toll gantries that are within a certain distance of the vehicle. Given the real-time requirements and the limited speed of the available hardware, a plain linear scan of the data would be infeasible even with about fifty objects. A real map of zones and gantries would hold orders of magnitude more objects than that (c.f., Table 3). This calls for some kind of geometric or spatial indexing [5], although the context places some heavy constraints on the data

This paper was presented at the NIK-2013 conference; see http://www.nik.no/.

structure used. One fundamental consideration is the complexity of the solution. In order to reduce the probability of errors, a simple data structure would be preferable. Beyond simplicity, and the need for high responsiveness, we have a rather limited, battery-driven piece of hardware to contend with.

The memory of the on-board unit is assumed to be primarily flash memory with serial access. The scenario is similar to that of a desktop computer, where the index would be stored on a hard drive, with a subset in RAM and the L2 cache. For an overview of the flash memory organization, see Fig. 1. The hierarchical memory architecture is summarized in Table 1. The serial nature of the memory forces us to read and write single bits at a time in a page. A read operation takes 50 µs. A write operation takes 1 ms, and may only alter a bit value of 1 to a bit value of 0. A sector, subsector or page may be erased in a single operation, filling it with 1-bits in approximately 500 ms. One important constraint is also that each page may typically only be erased a limited number of times (about 100 000), so it is crucial that our solution use the pages cyclically, rather than simply modifying the current ones in-place, to ensure wear leveling.

Based on the general needs of a self-contained tolling system, and on the hardware capabilities just described, we derive the following set of requirements:

A. The system must accommodate multiple versions of the same index structure at one time. The information may officially change at a given date, but the relevant data must be distributed to the units ahead of time.

B. It must be possible to distribute new versions as incremental updates, rather than a full replacement. This is crucial in order to reduce communication costs and data transfer times. It will also make it less problematic to apply minor corrections to the database.

C. The memory footprint must be low, as the memory available is highly limited.

D. The database must maintain 100 % integrity, even during updates, to ensure uninterrupted access. It must be possible to roll back failed updates without affecting the current database.

E. The indexing structure must be efficient in terms of CPU cycles during typical operations. Both available processing time and energy are highly limited and must not be wasted on, say, a linear scan over the data.

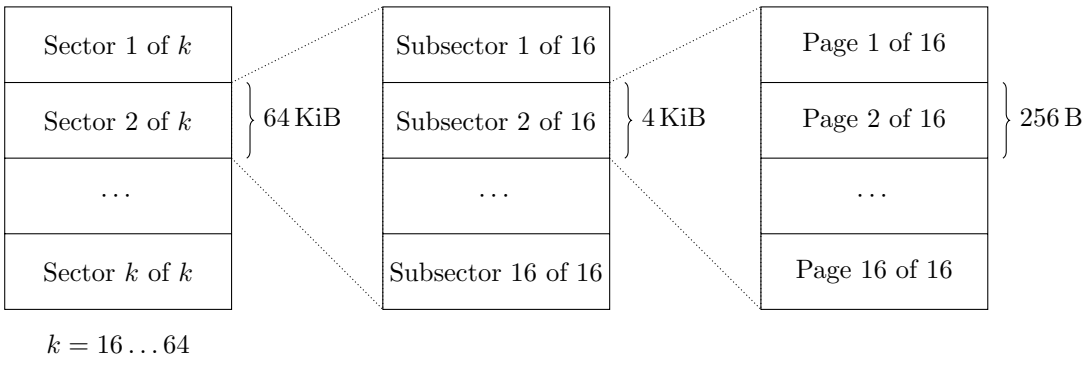


Figure 1: Flash memory architecture

Table 1: Hierarchical memory architecture

Description	Size	Access time	Persistent?
Processor internal memory	Ki-bytes	Zero wait state	No
External RAM	100 KiB	Non-zero wait states	No
Serial Flash	8, 16 or 32 MiB	See main text	Yes

F. The relevant data structure must minimize the number of flash page reads needed to perform typical operations, especially for search. Flash-page reads will be a limiting factor for some operations, so this is necessary in order to meet the real-time operation requirements.

G. In order to avoid overloading individual pages (see discussion of memory architecture, above), wear leveling must be ensured through cyclic use.

H. The typical operations that must be available under these constraints are basic two-dimensional geometric queries (finding nearby points or zones) as well as modification by adding and removing objects.

In addition, the system must accommodate multiple independent databases, such as tolling information for adjacent countries. Such databases can be downloaded on demand (e.g., when within a given distance of a border), cached, and deleted on a least-recently-used (LRU) basis, for example. We do not address this directly as a requirement, as it is covered by the need for a low memory footprint per database (Req. C).

As in most substantial lists of requirements, there are obvious synergies between some (e.g., Reqs. E and F) while some are orthogonal, and some seem to be in direct opposition to each other (e.g., Reqs. A and C). In Section 3, we describe a simple index structure that allows us to meet all our requirements to a satisfactory degree.

3 Our Solution: Immutable 9-by-9 Quadtrees

The solution to the indexing problem lies in combining two well-known technologies: quadtrees and immutable data structures.

In the field of geographic and geometric databases, one of the simplest and most well-known data structures is the quadtree, which is a two-dimensional extension of the basic binary search tree [see, e.g., 6, 7]. Just as a binary search tree partitions $\mathbb{R}$ into two halves, recursively, the quadtree partitions $\mathbb{R}^2$ into quadrants. A difference between the two is that where the binary search tree splits based on keys in the data set, the quadtree computes its quadrants from the geometry of the space itself.* There are several other spatial indexes out there [see, e.g., 5, 8]. We chose quadtrees mainly because of their simplicity. In addition, the lack of explicit keys in the internal nodes yields substantial space savings, which is crucial in our application.

The internal node size is fixed, regardless of the contents of sub-nodes. All objects stored in the quadtree have an extent from several meters to several kilometers. Some polygons will typically span entire countries. This fact, taken together with the

* Technically, this is the form of quadtrees known as PR Quadtrees [5, § 1.4.2.2].

scaling (see Table 2) and the high fanout (discussed below), leads us to a predefined maximum tree depth of 4 levels.

Objects stored in the tree often represent physical road infrastructure. There will typically be a natural spatial locality of physical objects that is reflected in a corresponding locality of reference in the tree structure (see Fig. 6). The objects will also tend to overlap several leaf regions, and possibly also the regions of higher-level nodes. Therefore, in its simplest implementation, the tree has a lot of duplicate pages. In our tests, the duplication has been as high as 65 %. These leaf nodes can easily be identified and the space reclaimed, if needed.

In order to reduce the number of node (i.e., page) accesses, at the expense of more coordinate computations, we increase the grid of our tree from 2-by-2 to 9-by-9. The specific choice of this grid size is motivated by the constraints of the system. We need 3 bytes to address a flash page and with a 9-by-9 grid, we can fit one node into $9 \cdot 9 \cdot 3 = 243$ bytes, which permits us to fit one node (along with some book-keeping data) into a 256 B flash page. This gives us a very shallow tree, with a fanout of 81, which reduces the number of flash page accesses considerably.

We return to how this data structure choice satisfies our requirements in Section 5, but here are some motivating advantages that it has over other, comparable structures:

- The structure is very simple. This, together with the copy-on-write semantics, is important in guarding against errors and making the system highly robust to failure.

- A perfectly balanced tree is less important than deterministic performance. With our quadtree we know there are at most 4 levels, so we need at most 4 flash page reads to get to a leaf node.

- The structure poses very few requirements on RAM. When inserting an object, we need to modify at most 4 pages, one for each level.

- The structure is well-suited for incremental updates. One can transfer a list of new objects to the in-vehicle unit, and it can update its tree with relatively limited computation, and little impact on the structure as a whole. As communication is done via GSM/GPRS, the bandwidth is highly limited. This means that performing a full update, transfering the full, updated tree from the server, is unfeasible.

The search is performed by a simple depth-first traversal of the nodes whose regions overlap with the query region. At the leaf node level, we may find that the the entire region is inside a given zone, in which case the overlap is a given. Otherwise, we need to perform actual polygon overlap computations [9]. All coordinate calculations and comparisons are performed in the integer domain after scaling the decimal latitude and longitude degrees by 1.0×10^7. This scaling creates an underflow when the square size reaches 1 cm^2. For an example of the resulting node sizes (area of ground covered), see Table 2. The last three columns show the number of leaf nodes at the various levels in the experimental build described in Section 4.

The use of latitudes and longitudes is well-behaved without singularities and ambiguities except at the poles and close to the date line, $\pm 180°$. Special care must be taken for databases spanning this line. Our design assumes that objects can be

sorted from west to east by increasing longitude, an assumption that is unwarranted in Russia and some smaller countries located on or close to the date line in the Pacific Ocean. This limitation may be mitigated by adding a fixed amount to the latitude to transform (by modular arithmetic) a range of, e.g., 175°E–175°W into 0°E–10°E. Another approach is to transform latitudes in the range 180°W–170°W into 180°E–200°E. This issue may apply to several parts of the system, including distance calculations, but a detailed treatment is beyond the scope of this paper.

Immutable data structures have been used in purely functional programming for decades [see, e.g., 10], and they have recently become more well known to the mainstream programming community through the data model used in, for example, the Git version control system [see, e.g., 11, p. 3]. The main idea is that instead of modifying a data structure in place, any nodes that would be affected by a given modification are duplicated. For a tree structure, this generally means the ancestor nodes of the one that is, say, added. Consider the example in Fig. 2. In Fig. 2(a), we see a tree consisting of nodes a through f, and we are about to insert g. Rather than adding a child to c, which is not permitted because of immutability, we duplicate the path up to the root, with the duplicated nodes getting the appropriate child-pointers, as shown in Fig. 2(b), where the duplicated nodes are highlighted. As can be seen, the old nodes (dotted) are still there, and if we treat the old a as the root, we still have access to the entire previous version of the tree.

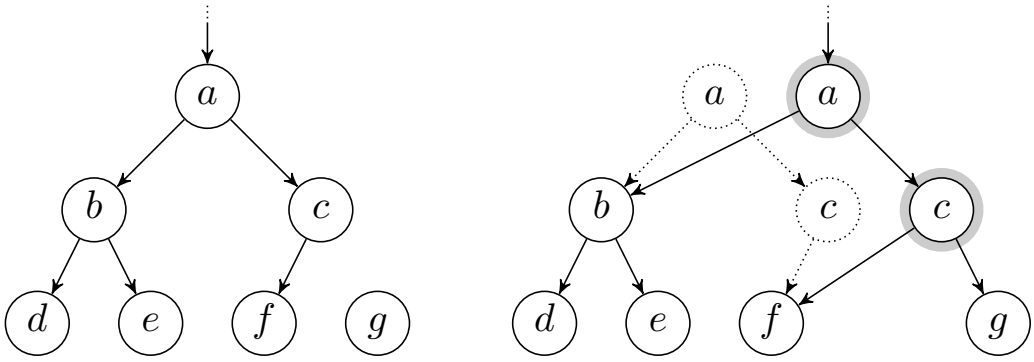
4 Experimental Evaluation

Our experiments were performed with a data set of approximately 30 000 virtual gantries (VGs; see Fig. 3(a)), generated from publicly available maps [12]. The maps describe the main roads of Norway with limited accuracy. Additionally, more detailed and accurate virtual gantries were created manually for some locations in the cities of Oslo and Trondheim. Fig. 4(a) shows the relevant virtual gantries in downtown Oslo, used in our test drive. There are about 35 virtual gantries on this route, and many of these are very close together. In general, there is one virtual gantry before every intersection.

Each local administrative unit (*kommune*) is present in the maps used [12], with fairly accurate and detailed boundaries. There are 446 such zones in total (see Fig. 3(b)). In addition, more detailed and accurate zones were created manually for some locations in Oslo and Trondheim. Some of these are quite small, very close together, and partially overlapping (see Fig. 4(b)).

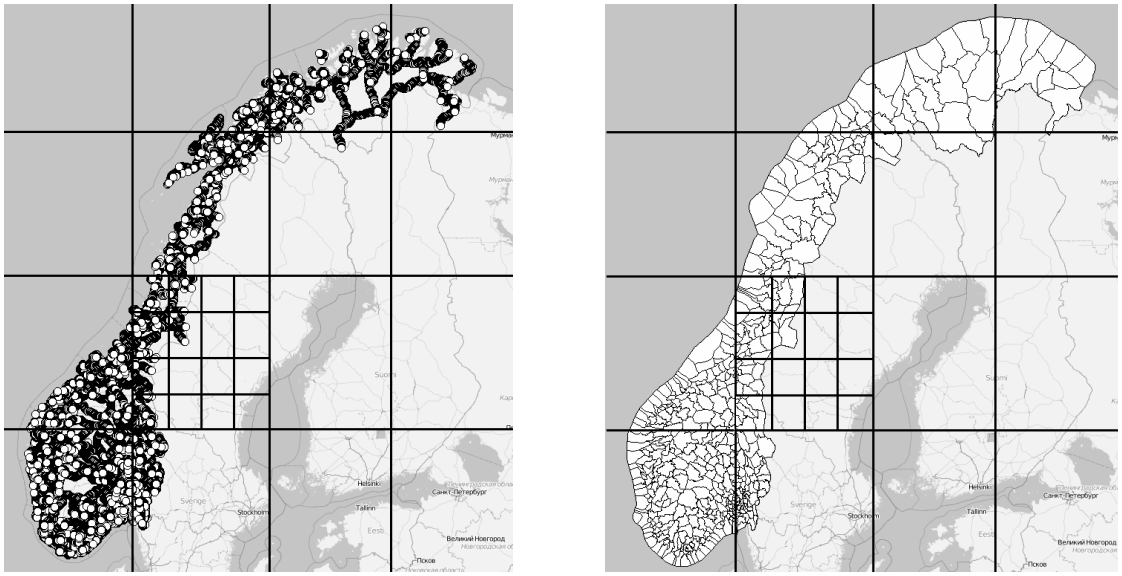
Table 2: Tree levels. Sizes are approximate

Level	**Size**	**Zone**	**VG**	**Both**
Top	2.0×10^6 m	0	0	0
1	2.1×10^5 m	15	0	4
2	2.4×10^4 m	625	168	471
3	2.5×10^3 m	81	12 353	39 721
4	3.0×10^2 m	0	157	486
5	3.0×10^1 m	0	0	0



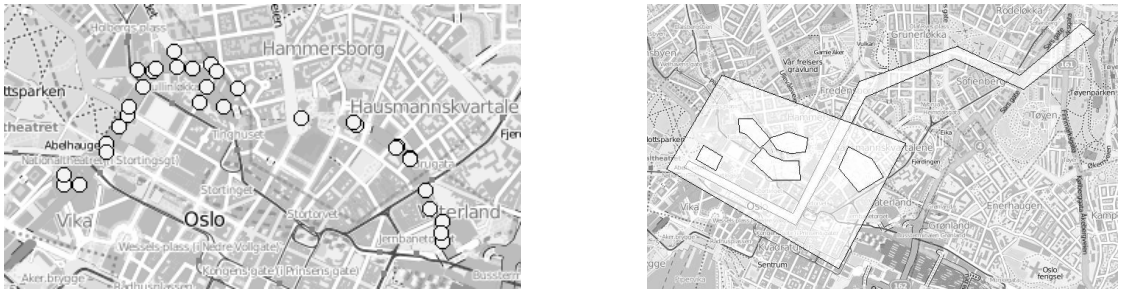
(a) Before insertion of g (b) After insertion of g

Figure 2: Node g is inserted by creating new versions of node a and c (highlighted), leaving the old ones (dotted) in place



(a) VGs (b) Zones

Figure 3: Virtual gantries and zones of Norway, with an illustrative quadtree grid



(a) VGs (b) Zones

Figure 4: Virtual gantries and overlapping zones in downtown Oslo

Data Structure Build

These test data were inserted into the quadtree as described in Section 3. Table 3 summarizes the important statistics of the resulting structure. The numbers for memory usage are also shown in Fig. 5, for easier comparison.

Judging from these numbers (row r), a database containing only zones would be quite small (about 0.38 MiB). Each zone is referenced by 5.5 leaf nodes on average (row f). Also note that only 57 leaf nodes (squares) are entirely contained in a zone (row j). This implies that the geometric inside/outside calculations will need to be performed in most cases.

This can be contrasted with the combined database of gantries and zones. The index is larger (3.82 MiB, row r), but the performance of polygon assessments is much better. Each polygon is referenced from 115 leaf nodes (row f) and there are more *inside* entries than *edge* entries (30 379 vs 21 333). This indicates that the geometric computations will be needed much less frequently.

Each of the three scenarios creates a number of duplicate leaf pages. Many leaf pages will contain the same zone edge/inside information. In our implementation of the algorithm, this issue is not addressed or optimized. It is, however, quite easy to introduce a reference-counting scheme or the like to eliminate duplicates, in this scenario saving 6 MiB (as shown in rows o through r).

The zone database contains very few empty leaf entries (row g), because the union of the regions covers the entire country, with empty regions found in the sea or in neighboring countries.

Flash Access in a Real-World Scenario

The index was also tested in a 2 km drive, eastbound on Ibsenringen, in downtown Oslo. The relevant virtual gantries are shown in the map in Fig. 4(a). The in-memory flash cache used was 15 pages ($15 \cdot 256$ B) with an LRU strategy, and the cache was invalidated before test start. Fig. 6 shows the result, in terms of flash accesses and the number of gantries found. At no point during the drive were more than 3 flash page reads required to complete the search. On average, 0.07 pages were read from flash each second. The numbers include only the index search—not the actual gantry definition.

5 Discussion

To view our results in the context of the initial problem, we revisit our requirement list from Section 2. We can break our solution into three main features: (*i*) The use of quadtrees for indexing; (*ii*) purely functional updates and immutability; and (*iii*) high fanout, with a 9-by-9 grid. Table 4 summarizes how these features, taken together, satisfy all our requirements. Each feature is either partly or fully relevant for any requirement it helps satisfy (indicated by ∘ or •, respectively).

Our starting-point is the need for spatial (two-dimensional) indexing (Req. H), and a desire for simplicity in our solution. The slowness of our hardware made a straightforward linear scan impossible, even with a data set of limited size. This led us to the use of quadtrees, whose primary function, seen in isolation, is satisfying Req. E, CPU efficiency. It also supports Req. C (low memory footprint) by giving us a platform for reducing duplication. Lastly, it supports efficiency in terms of

Table 3: Tree performance numbers

	Description	**Zones**	**VGs**	**Both**
a	Number of objects in database	448	29 037	29 485
b	Flash pages for index and data	2164	42 217	71 675
c	Size (MiB)	0.53	10.31	17.50
d	Flash pages for index	1716	13 180	42 190
e	Objects referenced by leaves	2481	43 263	95 272
f	Leaf nodes per object	5.5	1.5	115
g	Leaf entries not used, empty	240	27 403	1319
h	Leaf entries set	733	13 179	41 207
i	Max index tree depth	3	4	4
j	Zone inside entries	57		30 379
k	Zone edge entries	2406		21 333
l	Distinct leaf pages	578	11 608	14 138
m	Total leaf pages	721	12 678	40 682
n	Duplicate leaf pages	143	1070	26 544
o	Flash pages, dups removed	2021	41 147	45 131
p	Size, dups removed (MiB)	0.49	10.05	11.02
q	Index pages, dups removed	1573	12 110	15 646
r	Size of index, dups removed (MiB)	0.38	2.96	3.82

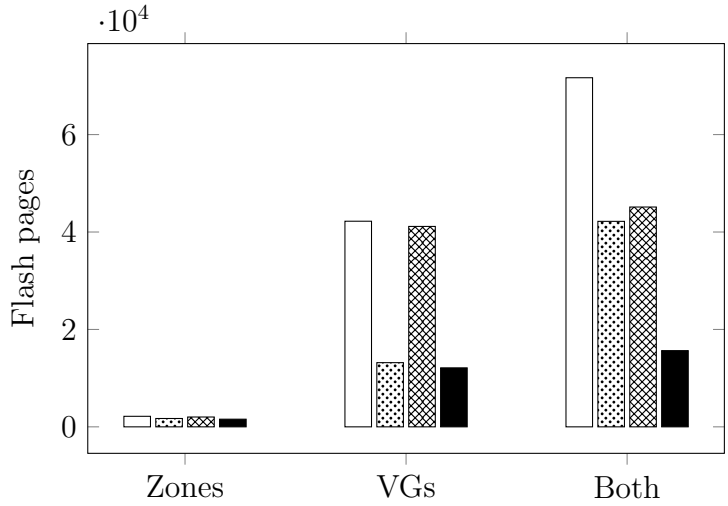


Figure 5: The plot shows total flash pages used ( ) and flash pages used for the index ( ), as well as the same with duplicates removed (and , respectively) for a data base consisting of zones, virtual gantries, or both (c.f., Table 3)

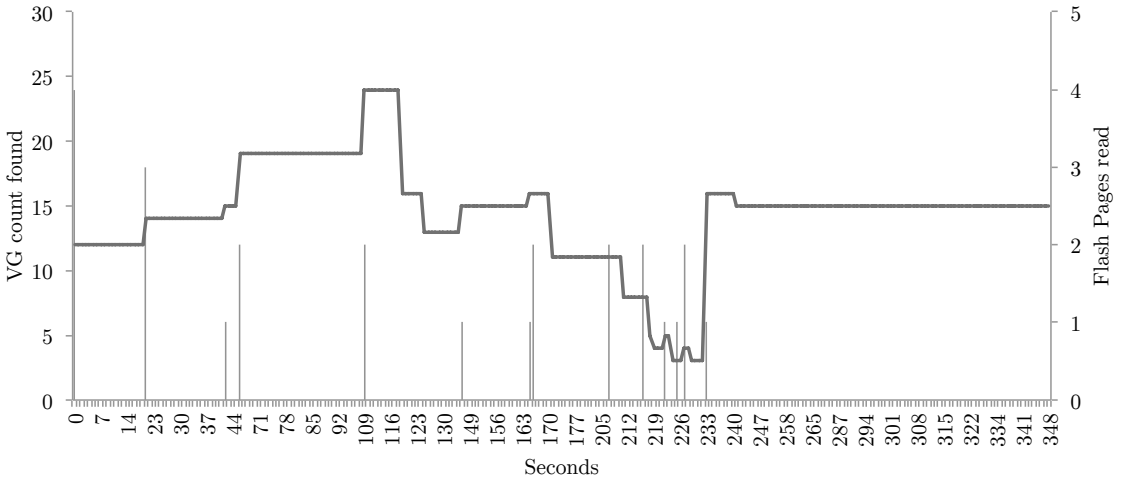


Figure 6: Flash access in an actual drive (Oslo Ring-1 Eastbound): flash pages read (vertical bars) and virtual gantries found (horizontal lines)

flash page accesses (Req. F), which is primarily handled by high fanout, using the nine-by-nine grid.

The purely functional updates, and the immutable nature of our structure, satisfy a slew of requirements by itself. Just as in modern version control systems such as Git [11], immutable tree structures where subtrees are shared between versions gives us a highly space-efficient way of distributing and storing multiple, incremental versions of the database (Reqs. A to C). This also gives us the ability to keep using the database during an update, and to roll back the update if an error occurs, without any impact on the database use, as the original database is not modified (Req. D). Finally, because modifications will always use new flash pages, we avoid excessive modifiations of any single page, and can schedule the list of free nodes to attain a high degree of wear leveling (Req. G).*

In our tests, as discussed in the previous section, we found that the solution satisfied our requirements not only conceptually, but also in actual operation. It can handle real-world data within real-world memory constraints (Table 3), and can serve up results in real time during actual operation, with relatively low flash access rates (Fig. 6).

Table 4: How components of the solution satisfy various requirements

Feature	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
The use of quadtrees for indexing			○		●	○		●
Purely functional updates and immutability	●	●	●	●			●	
High fanout, with 9-by-9 grid						●		

* For modifications that only entail adding new child nodes, a mutable structure would, of course, involve less writing in total, so this is not major argument for immutability. Also, a mutable structure would primarily yield lower wear leveling if the data in high-level nodes would need to be modified for each insertion.

6 Conclusion & Future Work

We have described the problem of real-time spatial indexing in an in-vehicle satellite navigation (GNSS) unit for the purposes of open-road tolling. From this problem we have elaborated a set of performance and functionality requirements. These requirements include issues that are not commonly found in indexing for ordinary computers, such as the need for wear-leveling over memory locations. By modifying the widely used quadtree data structure to use a higher fanout, and by making it immutable, using purely functional updates, we were able to satisfy our entire list of requirements. We also tested the solution empirically, on real-world data and in the real-world context of a vehicle run, and found that it performed satisfactorily.

Although our object of focus has been a rather limited family of hardware architectures, the simple, basic ideas of our index should be useful also for other devices and applications where real-time spatial indexing is required under somewhat similar flash memory conditions. Possible extensions of our work could be to test the method under different conditions, perhaps by developing a simulator for in-vehicle units with different architectures and parameters. This could be useful for choosing among different hardware solutions, as well as for tuning the index structure and database.

Our current implementation suffers from several limitations that might be addressed in future versions. For one thing, our leaf nodes have a fixed capacity (20 zones and 21 virtual gantries), with no overflow buckets. This could easily be solved by having a single overflow pointer in each leaf, pointing to a (possibly empty) linked list of overflow pages. Another issue is the problem of crossing 180° longitude. Some possible solutions are discussed in Section 3. Finally, some of our decisions are quite closely tied to the assumed memory architecture. Our use of 3 B block (flash page) pointers effectively limits the addressable flash memory to 4 GiB. However, modifying the pointer size as needed ought to be a trivial matter. Also, our specific tree design maps very well onto a 256 B flash page architecture, where individual pages may be addressed and erased individually. Many larger flash devices only support larger page sizes. This final limitation may be mitigated by using a sub-page addressing scheme. In its most naïve version, this will complicate the garbage collection procedure and waste memory. More advanced garbage collection procedures might further mitigate these problems, though.

Disclaimer & Acknowledgements Magnus Lie Hetland introduced the main design idea of using immutable, high-fanout quadtrees for the database structure. He wrote the majority of the text of the current paper, based in large part on the technical report of Lykkja [4]. Ola Martin Lykkja implemented and benchmarked the database structure and documented the experiments [4]. Neither author declares any conflicts of interest. Both authors have revised the paper and approved the final version. The authors would like to thank Hans Christian Bolstad and Erik Renno for fruitful discussions on the topic of the paper. This work has in part been financed by the Norwegian Research Council (BIA project no. 210545, “SAVE”).

References

[1] Frank Kelly. Road pricing: addressing congestion, pollution and the financing of britain's roads. *Ingenia*, 29:34–40, December 2006.

[2] Peter Hills and Phil Blythe. For whom the road tolls? *Ingenia*, 14:21–28, November 2002.

[3] Bern Grush. Road tolling isn't navigation. *European Journal of Navigation*, 6 (1), February 2008.

[4] Ola Martin Lykkja. SAVE tolling objects database design. Technical Report QFR01-207-1492 0.7, Q-Free ASA, Trondheim, Norway, 2012.

[5] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[6] Hanan Samet. Hierarchical spatial data structures. In *Design and Implementation of Large Spatial Databases. Proceedings of the First Symposium SSD '89*, volume 409 of *Lecture Notes in Computer Science*, pages 191–212. Springer Berlin Heidelberg, 1990.

[7] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*, chapter 14, pages 291–306. Springer-Verlag, 2nd revised edition, 2000.

[8] Mei-Kang Wu. On R-tree index structures and nearest neighbor queries. Master of science thesis, University of Houston, 2006.

[9] Hanan Samet and Robert Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):192–222, July 1985.

[10] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[11] Jon Loeliger and Matthew McCullough. *Version Control with Git*. O'Reilly, second edition, 2012.

[12] Kartverket. N2000 kartdata. Available from `http://www.kartverket.no/Kart/Kartdata/Vektorkart/N2000`, 2012.