A Spatial Hashtable Optimized for Mobile Storage on Smart Phones

Jörg Roth

Department of Computer Science
Univ. of Applied Sciences Nuremberg
Kesslerplatz 12
90489 Nuremberg, Germany
Joerg.Roth@Ohm-Hochschule.de

Abstract: This paper describes an approach to access large amounts of geo data on mobile devices with small runtime memories. We have this problem, e.g., if we want to execute a route planning application on smart phones that do not have server access. The *Spatial Hashtable* approach first orders all geo objects by their locality. This increases the probability to remain at the same hashtable position for subsequent queries and decreases the transfer from flash to runtime memory. Spatial Hashtables also provide spatial indexes that quickly allow an application to access entries by geometric conditions. We present three Spatial Hashtable mechanisms: *Pivot*, *Stripes* and *Double Stripes*.

1 Introduction

Mobile location-based applications such as tour guides often access spatial data via a mobile network on a central server. In some scenarios, however, it is reasonable to store a certain amount of spatial data on the mobile device. Reasons could be: a mobile network is not available in certain scenarios, it is too costly, or too slow. In addition, a certain business model could prefer not to install a service but to put the geo data on the mobile device.

Even though big mobile storages are nowadays available as flash memory (e.g. SD cards with 16-64 GB), the actual runtime storage for the heap in the internal memory is considerably small (e.g. 64 MB in current Android smart phones). Unfortunately, to load a block of data from flash memory takes (dependent on the real hardware) approx. 200 times longer compared to load a block from the internal memory.

For certain applications huge amounts of geo data are involved. For route planning in Germany, more than 2 million streets and over 10 million crossings have to be stored. The required space is approx. 1 GB. Thus, we have to think about a mechanism to load blocks of interest from the flash memory into the internal memory. In order to have a certain amount of related entries available in a specific block, we have to *localize* all geo data.

In this paper we introduce the notion of the *Spatial Hashtable* – a structure that combines a traditional hashtable with the ability to look up spatial data by geometric conditions. We present three approaches to localize the data.

2 Related Work

The problem has certain similarities to virtual memories in operating system [Vi01]. They also are based on a locality of objects. Whereas this type of locality means that related references have similar addresses, our locality means something different: objects with similar addresses describe areas in the real world that are spatially close together. Thus, virtual memory mechanisms can be used as underlying approach (especially swapping strategies), but we first have to achieve *spatial* locality of entries.

[MH97] assumes a grid with equal geographical tile size. Each tile contains the geo objects that reside in the respective geographical area, thus tiles do not contain equal numbers of entries. The approach strongly relies on virtual memory methods in the background. Once an entry inside a tile is accessed, the respective memory page is cached and thus access to proximate entries is faster. To improve the idea of grids, so called space-filling curves can order grid tiles in a specific ordering to preserve proximity [ARR+97]. However, grids always have the problem that geo data with an inhomogeneous geographical distribution (the usual case) lead to tiles with different number of entries.

Other approaches such as [ZXL02, HS03] focus on the replacement strategy for cache entries. They provide strategies to find out appropriate cache entries based on their geographic properties. For the special case of route planning there exist mechanisms that support a graph algorithm (usually a variation of A*) to access the routing network on the external memory [BB04]. Note that in this paper we do not focus on cache replacement strategies. We strongly believe that once geo data is spatially clustered, even non-spatial cache replacement strategies highly perform.

The Spatial Hashtable approach presented in this paper is based on ongoing research on geo data. *HomeRun* [Ro10a, Ro10b] is a platform for low-cost development of location-based services, especially of small services outside the mass market. HomeRun provides a set of basic services, e.g. the import of geo data from public sources. It contains an efficient communication infrastructure for service usage and offers functional blocks for mobile devices, e.g. to display maps. As basic functions already exist, a service developer can concentrate on the actual application. A development based on HomeRun is cost-effective and due to the modularization, can easily be modified later. As a major direction, we want to provide location-based services on smart phones (online and offline). In [Ro11] we presented a first approach to store and access huge amounts of geo data on smart phones. It was based on a relational database that used a special spatial indexing mechanism [Ro09]. It turned out that, even though relational databases are a convenient way to access geo data (even on smart phones), we suffer from low performance. For applications such as route planning, relational databases are not appropriate.

3 The Spatial Hashtable

3.1 Basic Considerations

Our Spatial Hashtable is based on the following assumptions:

- The spatial data is completely mastered on a development system that exports an optimized structure for the mobile device. The mobile device mainly reads spatial data rather than changing it.
- On the mobile device, the transfer between flash memory and runtime heap is considered as the major performance bottleneck. Once inside the heap, spatial data can be processed quickly, as current smart phones have fast CPUs.
- We expect a mobile application to perform queries with spatial locality, i.e. successive queries usually affect data of the same geographic region.
- Data entries can be loaded using two mechanisms. First: the application can use an index to address an entry, similar to an array index. Second, an application can formulate a certain spatial condition such as: all entries that are located inside a certain geographic rectangle.
- Additionally, we assume that all entries have the same structure with fixed fields, i.e. the Spatial Hashtable can be compared to a relational database table.

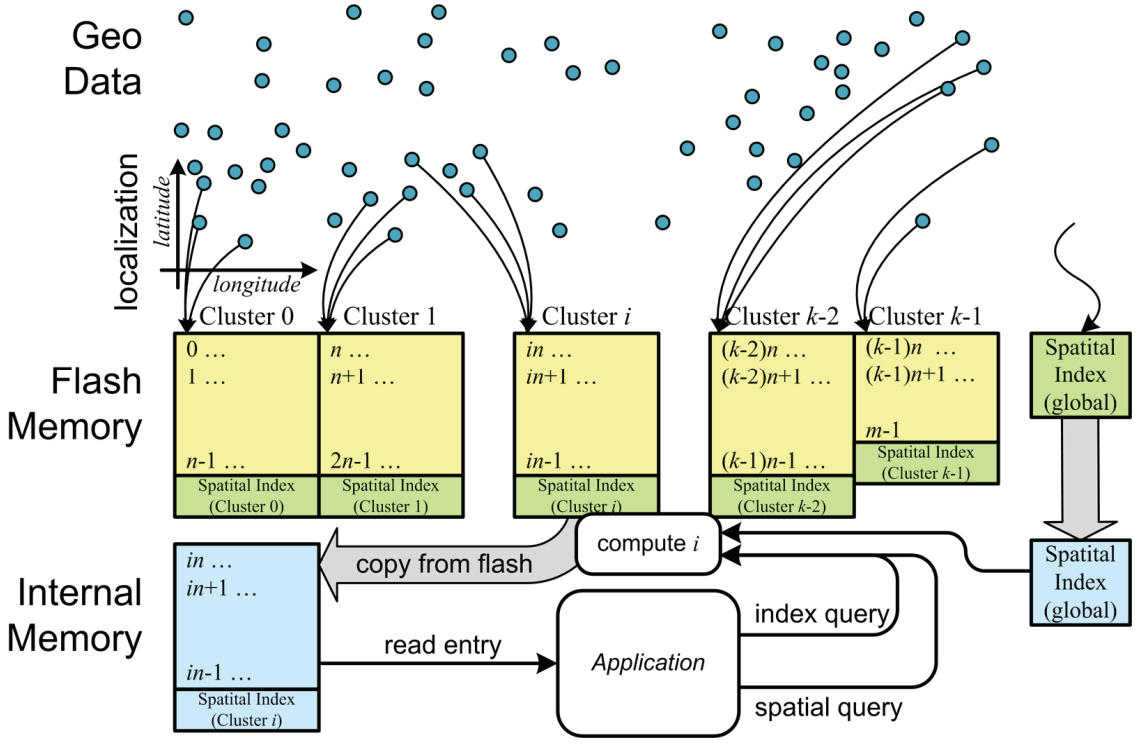

Figure 1: Basic Spatial Hashtable idea

Fig. 1 illustrates the Spatial Hashtable idea. We assume an amount of m geo data entries. We divide the overall spatial data into k clusters that are stored in separate files in the flash memory. The number of entries per cluster n is constant except for the last cluster that gets the rest of m entries. The mapping of the geo data entries to the specific cluster file is called *localization*. This step is essential for the later runtime performance. We present three localization mechanisms *Pivot*, *Stripes* and *Double Stripes* in more detail later.

One cluster file (or more if we use caching) has to fit into the smart phone's heap memory. The application can load cluster files via two mechanisms: first, it can access a geo data entry by its index. For an index x, the cluster file that contains this entry has the number $i=\lfloor x/n \rfloor$. The second mechanism provides geometric queries. For this, spatial indexes are written to flash memory during the localization procedure. A *global* spatial index is used to identify clusters that fulfill certain geometric conditions. Spatial indexes for each cluster are used to spatially search entries in a cluster.

Note that the loading and accessing mechanism is shielded behind an API. The application simply formulates a query for a certain entry – the runtime system executes the computation of the specific cluster index and loads it into memory transparently. Further note that in real systems, we allow multiple clusters to reside in the internal memory. This dramatically increases the hit rate. Note that, as the clusters are already localized, we can use simple cache replacement strategies such as LRU (Least Recently Used) that do *not* consider the spatiality of entries.

Localizing Geo Data

The main objective of the localizing algorithm is to define geographical regions with identical numbers of entries that divide the overall space without overlap. The probability for multiple requests to remain in the same cluster file should be high. Fig. 2 illustrates the problem.

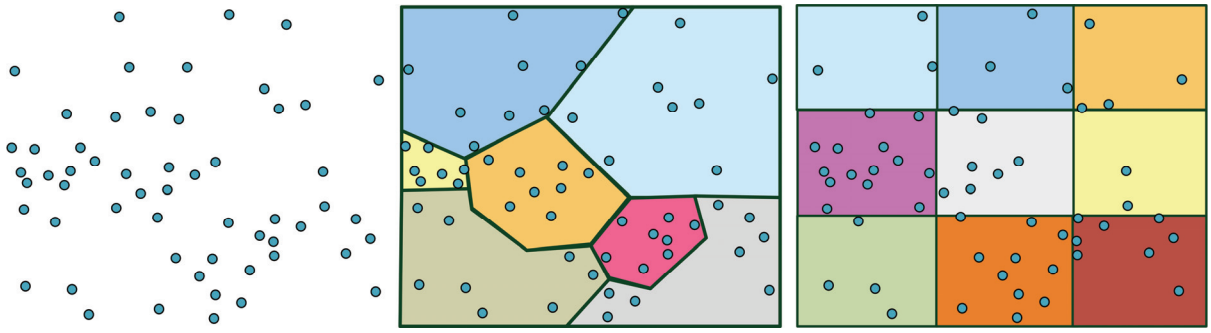

Figure 2: The localization problem

All geo data entries (fig. 2 left) are distributed in a two-dimensional space. Usually we have different densities, e.g. a higher number of entries in cities compared to the countryside. Fig. 2 middle shows a segmentation of the space into clusters with equal numbers of entries. Obviously, there exist different ways to divide the space into clusters. Ideal clusters have similar spatial heights and widths, as the probability for nearby entries to reside in the same cluster is higher.

We require equal cluster sizes (in terms of record numbers, *not* area sizes) as we want to increase the usage of the spare internal memory of the smart phone. If cluster sizes were not equal, the reserved space for geo data is usually not fully used. Thus a simple tile grid (fig. 2 right) is not suitable for our approach, as only the spatial sizes would be equal, but not the number of entries.

In the remaining paper we use the German road map topology as example geo data to show the Spatial Hashtable mechanisms. Table 1 shows the corresponding numbers of entries.

Table 1: Typical amounts of data for route planning (Germany)

	Records in million	**Bytes per record**	**Total size in MB**
Crossings	4.7	84	374
Links between crossings	10.9	27	280
Complete roads including names	2.3	139	310
		Sum	964

In the following, we focus on the localization mechanism and present three approaches.

3.2 Pivot Localization

The idea behind the Pivot localization is to build a cluster around a selected point called *Pivot* point. After the Pivot point and n-1 nearest points are clustered, a next Pivot point is selected and so forth.

The pseudo code of the Pivot localization is as follows:

1. Select a Pivot entry of all unselected entries (see below).
2. Sort all unselected entries by their distance to the Pivot entry.
3. Select n-1 nearest entries (or less for the last cluster). The Pivot entry and these n-1 entries form a new cluster.
4. Continue with (1) until all entries are selected.

Fig. 3 illustrates the approach.

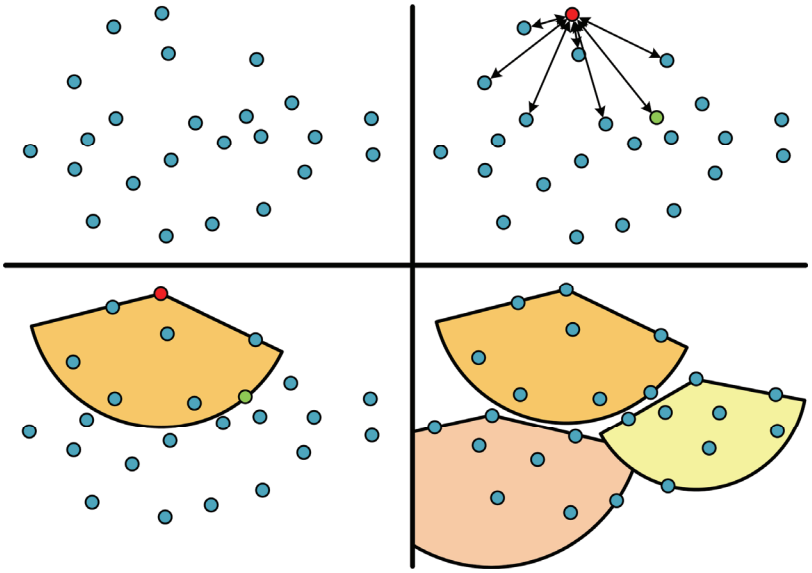

Figure 3: The Pivot localization

The selection of the Pivot point is important to form compact clusters. Especially the last clusters often get large and segmented if we used inappropriate Pivot points. There can be different ways to select the Pivot point, e.g.

- select the most northern unselected point;
- first compute the geometric centre of all points. For each round select the unselected point nearest to the centre.

It turned out that selecting the most northern point is an appropriate and simple to compute approach. Experiments show that the segmentation of the last clusters is less than in other approaches. Fig. 4 shows localization results of all German crossings (cluster size 800 kB).

During the localization procedure we store the coordinates p_i of the Pivot entries and the maximum distances r_i between all cluster entries to the Pivot entry. We store this list in the runtime memory, to quickly look up the appropriate cluster file. To find the cluster that contains a certain position q we proceed as follows:

1. Iterate through the list (p_i, r_i)
2. If distance(q, p_i) $\leq r_i$: stop with success, i is the cluster file
3. If the list is processed without any hit: stop with failure

Here, distance() computes the spatial distance between two coordinates. Note that this code assumes that q is actually available as entry. Usually, we perform a radius search, i.e. we want to look up all cluster files that contain entries inside a certain distance s to the given point q:

1. Iterate through the list (p_i, r_i)
2. If distance(q, p_i) $\leq r_i+s$: i is *one* cluster file (continue loop)
3. Process the entire list, the result is a list of selected cluster files.

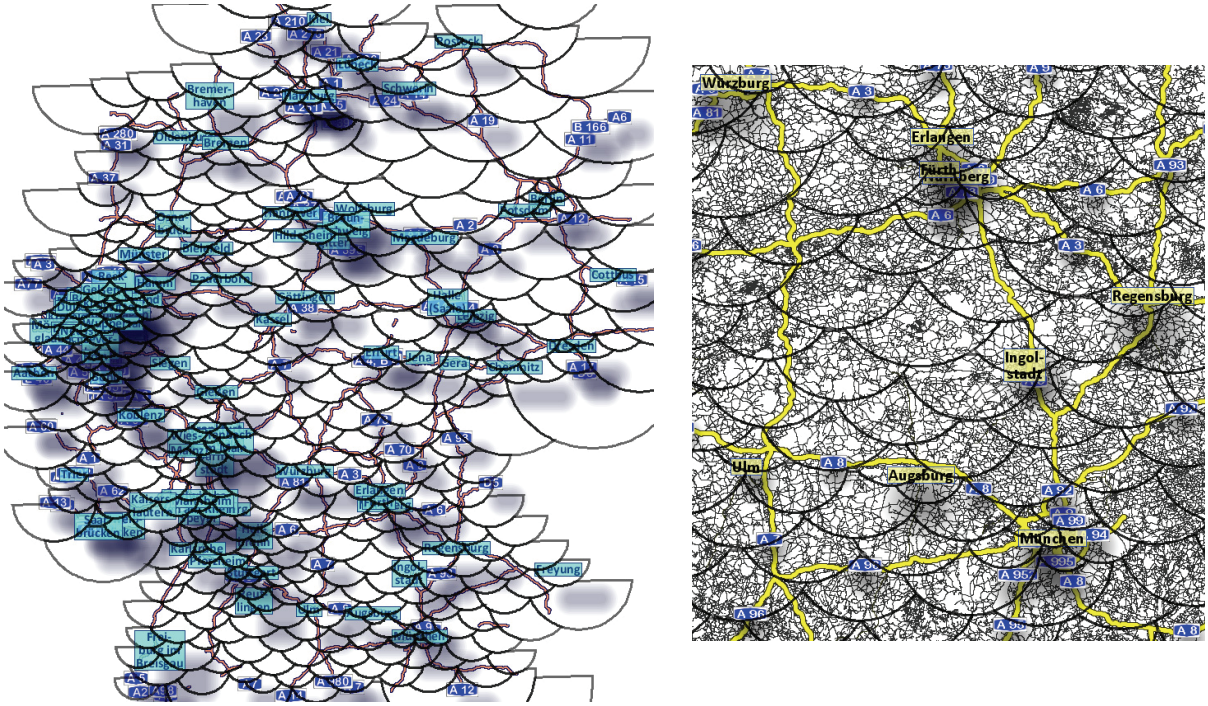

Figure 4: Pivot localization results

Note that this approach sometimes selects clusters that geometrically cannot overlap with the search area. This is because the actual cluster shapes are not circles; instead, they are circles where former cluster shapes are subtracted. An exact computation of the geometric condition would be computationally too expensive. However, if we use the most northern entry as Pivot entry, we can formulate a further condition to step (2):

$$\text{distance}((p_{i,lat}, q_{long}), (q_{lat}, q_{long})) \leq s$$

(Note that the $(p_{i,lat}, q_{long})$ is not a write error.)

In reality we do not linearly iterate through the list of clusters but use a spatial index (see section 3.5). This pseudo code is only to illustrate the mechanisms.

3.3 Stripes Localization

The Stripes localization follows another direction to form clusters. Only rectangular clusters are created. The benefit: spatial indexing is simplified as most indexing mechanisms are optimized for rectangular shapes.

As earlier mentioned, a simple grid is not appropriate as the number of entries per grid depends on the geometrical distribution density. Thus our idea is to first consider geometrical stripes that contain multiplies of n entries. These stripes are divided vertically with clusters of n entries. To get optimal clusters, we test multiple stripe widths and measure the quality of the resulting clusters using a function q.

The pseudo code of the Stripes localization is as follows:

1. Order all entries west to east.
2. Let u denote the number of not-clustered entries. Iterate i from 1 to $\lceil u/n \rceil$.
3. Create a meridian stripe of $\min(i \cdot n, u)$ entries starting from the west border of non-clustered entries.
4. Create clusters of n entries (of less for the last cluster) selecting entries from north to south.

5. For each cluster, compute the value q. Let the overall stripe value $q_i=\sum q/i$.
6. Continue iterating through i. Let i_{min} be the stripe index with the minimal value q_i.
7. Create clusters for i_{min} as tested in steps (3) and (4).
8. Continue with (2) until all entries are selected.

Fig. 5 illustrates the approach.

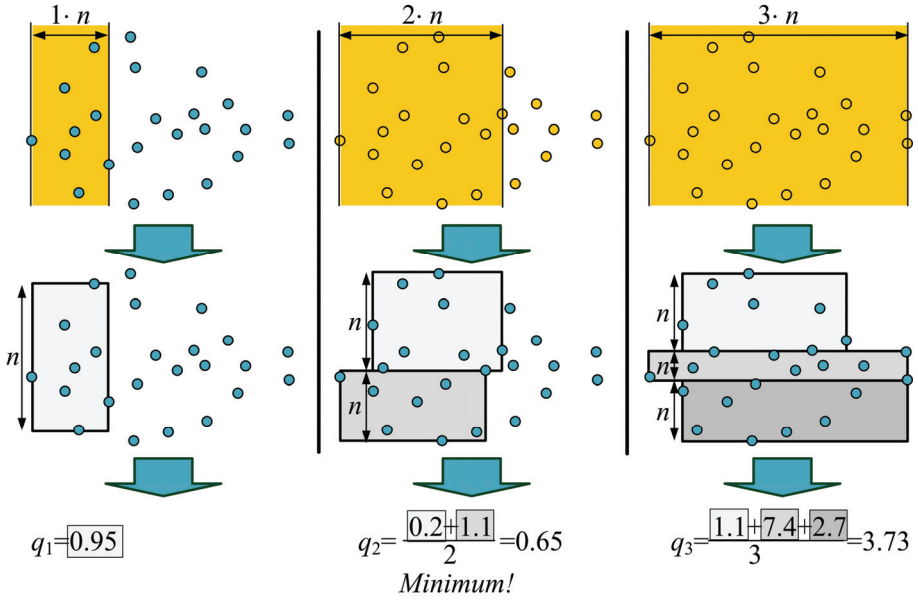

Figure 5: The Stripes localization

The function q should give low values for 'good' cluster shapes. The value for squares should be 0. In addition, turned rectangles (by 90°) should return the same value. We define q as follows:

$$q(\Delta_{lat},\Delta_{long}) = \begin{cases} \dfrac{\Delta_{lat}}{\Delta_{long}} - 1 & \text{if } \Delta_{lat} > \Delta_{long} \\ \dfrac{\Delta_{long}}{\Delta_{lat}} - 1 & \text{otherwise} \end{cases}$$

for clusters with geographical extent Δ_{lat} and Δ_{long} (see fig. 6).

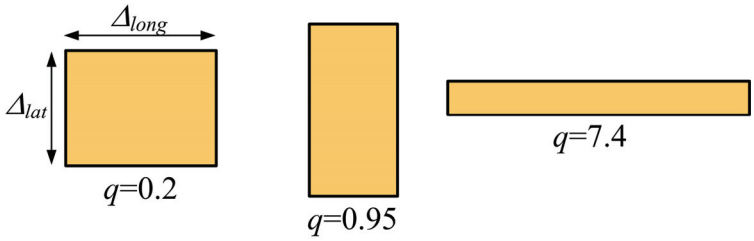

Figure 6: q value examples

Fig. 7 shows localization results of all German crossings (cluster size 800 kB).

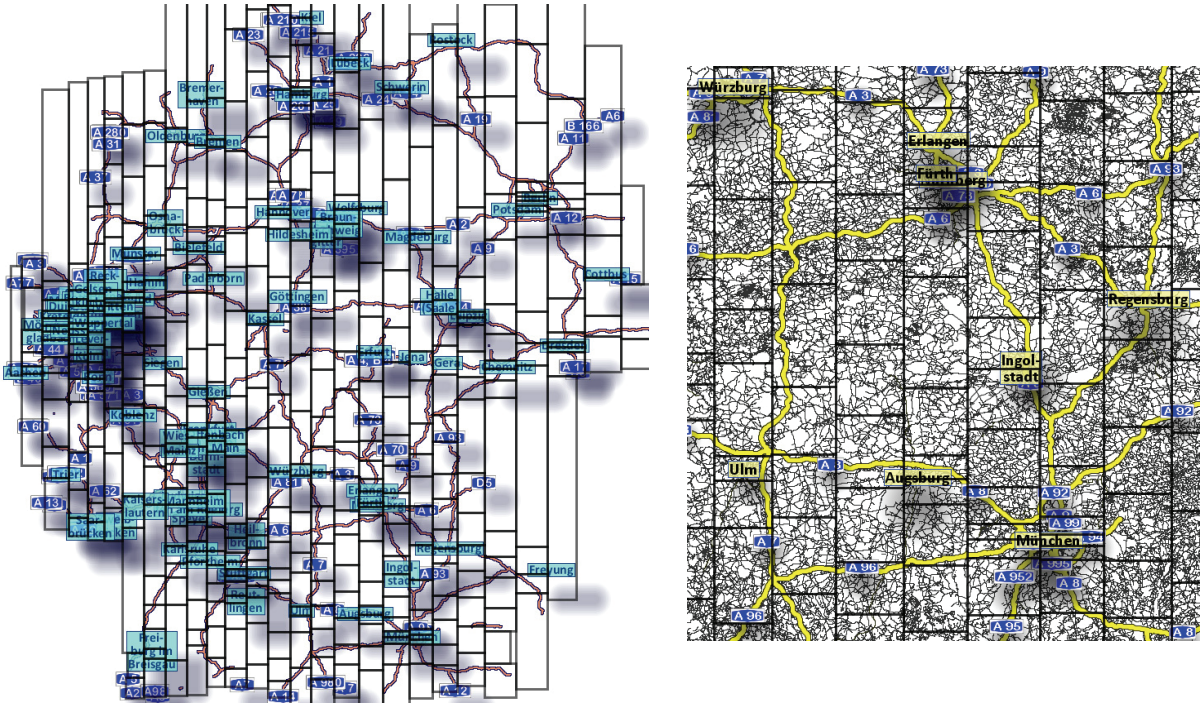

Figure 7: Stripes localization results

For spatial indexing, the localization mechanism collects the cluster's border ($north_i$, $south_i$, $west_i$, $east_i$). We perform these steps to look up all cluster files that contain entries inside a certain distance s to the given point q:

1. Iterate through the list ($north_i$, $south_i$, $west_i$, $east_i$)

2. If $q_{lat} \oplus s \leq north_i$ AND $q_{lat} \oplus -s \geq south_i$ AND $q_{long} \oplus s \geq west_i$ AND $q_{lat} \oplus -s \leq east_i$ then i is one cluster file (continue loop)

3. Process the entire list, the result is a list of selected cluster files.

Here, $\oplus$ denotes a function that moves a latitude or longitude value a certain amount of meters. Again note that in reality, we do not iterate through the list of clusters but use a spatial index (see section 3.5).

3.4 Double Stripes Localization

The Stripes localization has a certain disadvantage: stripes that both cover zones with high and low densities result in inhomogeneous cluster shapes: clusters that cover high densities tend to horizontal bars, whereas clusters that cover low densities tend to vertical bars. This is because a stripe defines the horizontal extent of all clusters inside the stripe. Unfortunately, typical geo databases both contain high density zones (e.g. cities) and low density zones (countryside).

The Double Stripes localization addresses this problem: stripes can be both divided vertically and horizontally. Fig. 8 illustrates problem. For a given distribution the Stripes localization generates horizontal bars for high density zones (left). Dividing the lower zone horizontally would result in a more suitable solution (right).

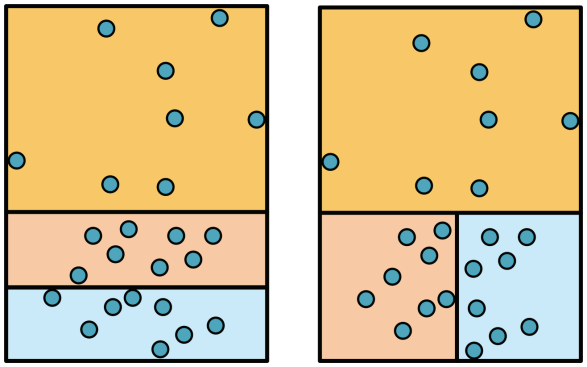

Figure 8: The problem of the Stripes localization

The Double Stripes localization uses the Stripes algorithm but replaces step (4). Here, we do not only test a single partitioning from north to south but test all permutations of horizontal and vertical partitions as illustrated in fig. 9.

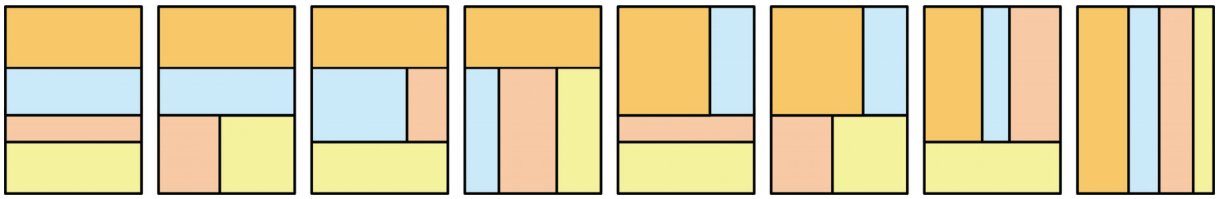

Figure 9: Eight possible ways to create four clusters for a given stripe

For each permutation we again compute the value q and look for the overall minimum.

Some considerations to the computational costs: For stripes with i clusters, there exists 2^{i-1} variations to partition a stripe. For stripe sizes larger than 25 clusters, it is too expensive to check all variations. Thus it is reasonable to formulate the loop in step (2): Iterate i from 1 to $\min(\lceil m/n \rceil, c)$, where c is a constant that limits the stripe size.

To reduce the number of variations even more, we additionally can limit the number of horizontal partitions by a number λ. For e.g., λ=3 we only allow 1, 2 or 3 horizontal partitions. For λ=2, the total number of variations for i clusters is the Fibonacci number F_{i+1}. For large numbers we can use the approximation

$$F_{i+1} \approx \frac{5+3\sqrt{5}}{10} \cdot \left(\frac{1+\sqrt{5}}{2} \right)^{i-1} \approx 1.171 \cdot 1.618^{i-1}$$

which is far lower than 2^{i-1}. However for larger values of λ, the benefit gets lower. Note that for λ=1 the Double Stripes approach produces the Stripes approach, thus Double Stripes could be considered as the more general approach.

For large stripes, however, the number of permutations still is far too large. Thus, we consider a third type of limitation that is based on the following observation: if we got an optimal partitioning of a stripe, every horizontal cluster border marks two optimal partitionings – one north of, one south of the border. Thus, we only have to "guess" an optimal border between north and south and we can check each part individually then. To check all borders recursively would lead to the same number of permutations. We thus use the following approach: Let μ be the maximum stripe size that is fully checked and Π be a set of numbers, each of it less than μ. We define a recursive function *getBest*:

```
getBest(from, count)
    if (count≤μ)
        return the best of all permutations of clusters (from… from+count-1)
    else
        for each j∈Π
            getBest(from, j)
            getBest(from+j, count-j)
            compute joined value of the two results
        return the best of the joined values
```

With the help of this approach, we can reduce the number of permutations that have to be checked from several billion to some thousands. Fig. 10 shows localization results of all German crossings with $\lambda=4$, $c=50$; $\mu=10$; $\Pi=\{5, 7\}$, cluster size 800 kB.

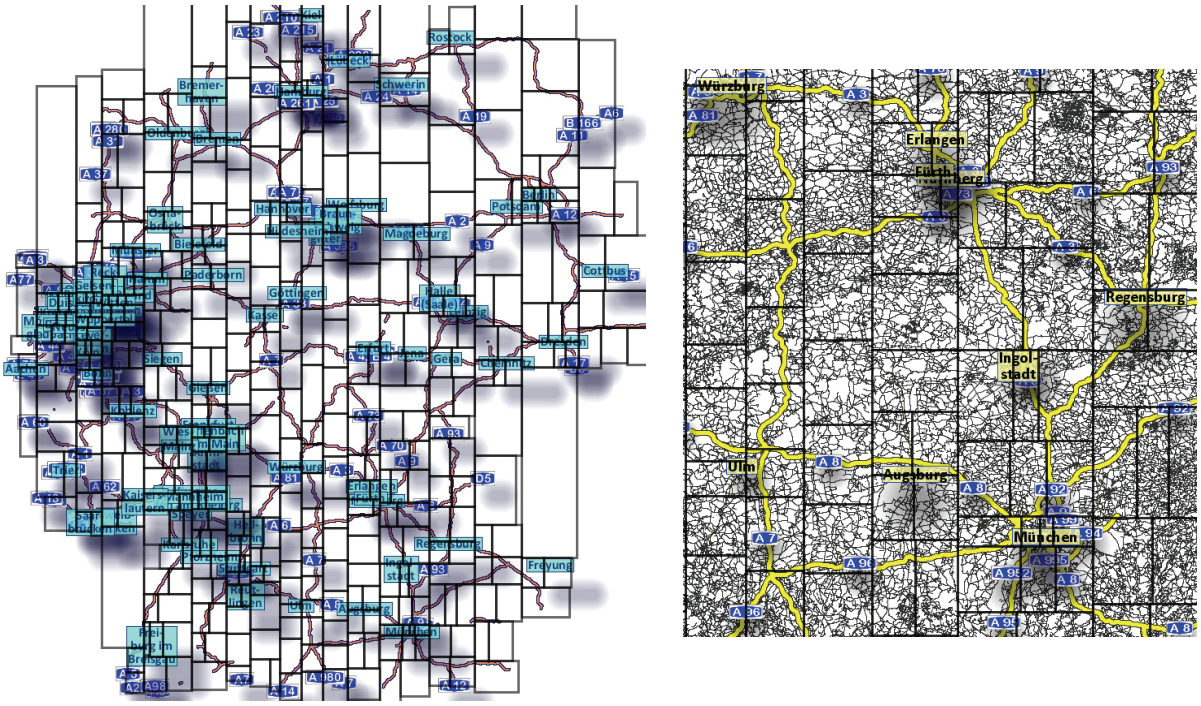

Figure 10: Double Stripes localization results

This approach, although computationally expensive, significantly improves the cluster shape. Table 2 shows a comparison of q values for the Germany data.

Table 2: Comparison of average q values

Localization	Cluster size 800 kB	Cluster size 400 kB
Stripes	1.182	1.371
Double Stripes	0.449	0.456

As a major result, Double Stripes rectangles are more square-like with a ratio of approx. 1.45 between the two lengths.

3.5 Spatial Indexing

To get a specific geo entry by its geometry we have to perform two steps:

- Identify the cluster file that fulfils the geometric condition.
- Read the entry inside the cluster file that fulfils the geometric condition.

Note that, dependent on the specific geometric condition, more than one cluster file and multiple entries may fulfill the condition. As multiple cluster files may not fit into the runtime memory, we assume that the application iterates through the result set in a cursor-based manner.

We speed up both lookup steps using an R-Tree spatial index [Gut84]. The *Gobal Spatial Index* (fig. 1) contains the borders of the cluster files. Separate spatial indexes inside the clusters provide access the specific entries. Note that we can pre-compute these R-Trees. This especially means that they can be balanced in order to decrease the average access time. R-Trees are typically used in a dynamic environment where geo data may change and the R-Trees must be balanced 'on the fly'. In contrast, we can spend time to balance the trees offline.

R-Trees use rectangular areas to approximate geometries. Thus, Stripes and Double Stripes are suitable approaches as they already provide rectangular cluster shapes. In contrast Pivot cluster areas may cause additional lookup time as the actual geometry has to be checked.

4 Evaluation

We want to investigate the benefit of the localization methods. An application that processes geo data should work as long as possible with a specific cluster file. We made two analyses:

1. How high is the probability for two entries in a certain spatial distance to reside in the same cluster file? This value can also be considered as a hit rate for cluster files.
2. In the street network: if two crossings are connected by a street – how high is the probability that these crossings reside in the same cluster file? This analysis is strongly related to the route planning application: following streets from crossing to crossing is a basic operation inside the path finding algorithm.

Besides our three localization methods we integrated another method as a comparison reference. We call it *Northsouth*: we just order all entries from north to south. This method randomizes the spatial relation as east to west proximity is destroyed. The Northsouth method thus represents a 'no localization' approach.

4.1 Proximity Hit Rate

We created two Spatial Hashtables from the 4.7 million crossings of Germany with a total size of 374 MB. We checked with two cluster sizes 400 kB and 800 kB (table 3).

Table 3: Cluster properties for different cluster sizes

	Cluster size 400 kB	**Cluster size 800 kB**
Entries per cluster	4876	9752
Number of clusters	959	480

Fig. 11 represents the outcome of the first analysis. For two pairs of entries with a certain distance we checked if they are in the same cluster file. We could also take this view: if an application loaded a certain entry and wants to load another entry in a certain distance: what is the expected hit rate to remain in the same cluster file?

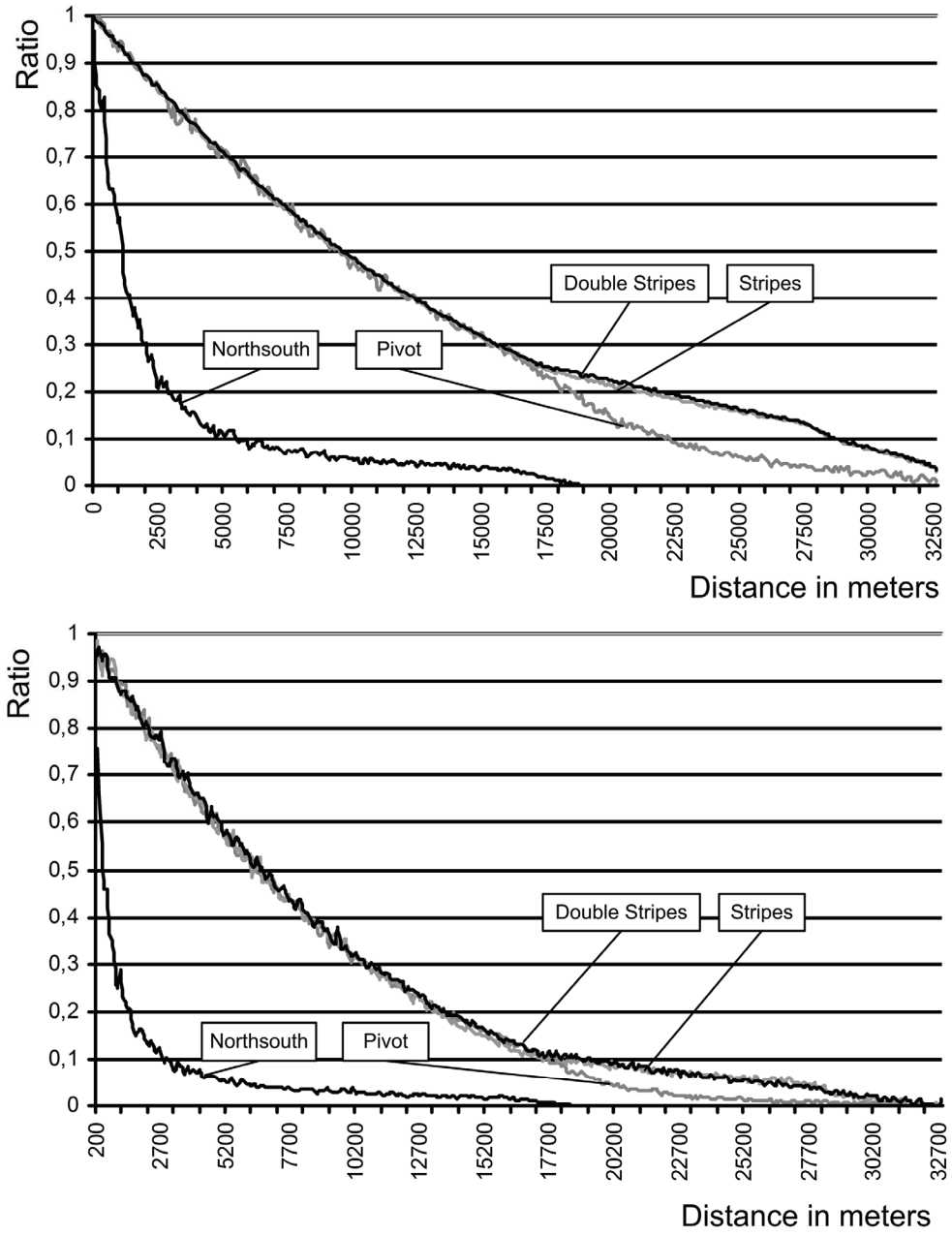

Figure 11: Ratio of two entries in a certain distance that are in the same cluster file
(cluster size 800 kB top, 400 kB bottom)

The main observations:

- The three real localizations are much better than the Northsouth localization.
- Pivot, Stripes and Double Stripes only slightly differ. Beyond a certain distance, Pivot is less effective. Stripes and Double Stripes provide virtually the same results.

Even though this analysis justifies the localization as such, the specific method does not seem to have significant influence on the effectiveness. This is a disappointing result, especially if we consider the complexity of Double Stripes.

4.2 Effectiveness Considering a Certain Application

We could argue that the first evaluation was set in an artificial scenario as it does not consider real application requests. In the second analysis we use a route planning application [Ro12] that accesses a street network of crossings and streets. In the underlying A* algorithm, the basic operation is to follow a street segment from a given crossing to the next crossing. It is important that we have a high probability to remain in the same cluster file. Table 4 represent the 'inverse' case: how many connected crossings are in different cluster files?

Table 4: Ratio of connected crossings that are in different cluster files

	% Ratio (avg. all clusters)		% Ratio (worst cluster)	
Cluster size	400 kB	800 kB	400 kB	800 kB
Pivot	1.09	0.73	1.99	1.46
Stripes	1.10	0.75	2.54	1.37
Double Stripes	1.07	0.73	1.84	1.19
Northsouth	10.43	5.39	16.31	8.44

Again, the difference to Northsouth is significant, whereas the differences between Pivot, Stripes and Double Stripes are low. Double Stripes is slightly better than the others.

If we take into account these analyses and the different properties, we can sum up:

- Localization is a useful tool to store large amounts of geo data.
- The runtime complexity of Double Stripes cannot be justified in comparison to Stripes. Even though Double Stripes constructs more square-like clusters, the benefit does not pay off.
- Pivot localization is easy to implement and provides good results. Its only drawback is non-rectangular cluster regions, which may be difficult for spatial indexing.

We used the Double Stripes localization with 800 kB clusters to implement the *donavio* routing planning [Ro12]. All required routing data for Germany, i.e. the street topology, street geometries, names, and driving properties such speed limits allocate a storage space of approx. 1.3 GB on the smart phone's SD card. The runtime heap in contrast only has a size of max. 64 MB. As the routing algorithm A* usually performs several queries within the same region, our approach is suitable. It takes only some seconds to compute a fastest route from point to point.

5 Conclusions

We presented the *Spatial Hashtable* approach and introduced three mechanisms *Pivot*, *Stripes* and *Double Stripes* to localize geo data. It turned out that localization is an important mechanism to ensure performance even on small systems like smart phones.

Currently, we spatially represent geo data entries by a single point (i.e. its geometric center). Larger objects such as city borders would cut multiple cluster file borders, thus the existing approach is not suitable. In the future we work on an extended version of Spatial Hashtables that also can hold such objects.

References

[ARR+97] Asano, T.; Ranjan, D.; Roos, T.; Welzl, E.; Widmayer, P.: Space-filling curves and their use in the design of geometric data structures, Theoretical Computer Science, Vol. 181, Issue 1, 1997, 3–15

[BB04] Breunig, M.; Baer, W.: Database support for mobile route planning systems, Computers, Environment and Urban Systems, Vol. 28, Issue 6, 2004, 595–610

[Gut84] Guttman A.: R-Trees: A Dynamic Index Structure for Spatial Searching, Proc. of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 1984, 47-57

[HS03] Höpfner, H.; Sattler, K. U.: Towards Trie-Based Query Caching in Mobile DBS, in proceedings of the Workshop Scalability, Persistence, Transactions- Database Mechanisms for Mobile Applications, Lecture Notes in Informatics (LNI), 2003, 106-121

[MH97] McCormack, J. E.; Hoog, J.: Virtual Memory Tiling for Spatial Data Handling in GIS, Computer & Geosciences Vol. 23, No. 6, 1997, 659-669

[Ro09] Roth, J.: The Extended Split Index to Efficiently Store and Retrieve Spatial Data With Standard Databases, IADIS International Conference Applied Computing 2009, Rom (Italy), 19.-21. Nov. 2009, Vol. I, 85-92

[Ro10a] Roth, J.: Die HomeRun-Plattform für ortsbezogene Dienste außerhalb des Massenmarktes, in Zipf A., Lanig S., Bauer M. (Hrsg.) 6. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", Heidelberger Geographische Bausteine Heft 18, 2010, 1-9

[Ro10b] Roth, J.: Übernahme von Geodatenbeständen aus Open Street Map und Bereitstellung einer effizienten Zugriffsmöglichkeit für ortsbezogene Dienste, Praxis der Informationsverarbeitung und Kommunikation (PIK), Vol. 13, Heft 4, 2010, 268-277

[Ro11] Roth, J.: Moving Geo Databases to Smart Phones – An Approach for Offline Location-based Applications, Innovative Internet Computing Systems (I2CS), Berlin, 15.-17. Juni 2011, GI Lecture Notes in Informatics, Vol. P-186, 228-238

[Ro12] Roth, J.: Modularisierte Routenplanung mit der donavio-Umgebung, 9. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", 2012

[Vi01] Vitter, J. S.: External memory algorithms and data structures: dealing with massive data, ACM Computing Surveys (CSUR), Vol. 33 Issue 2, June 2001, 209-271

[ZXL02] Zheng, B.; Xu, J.; Lee, D. L.: Cache Invalidation and Replacement Strategies for Location-Dependent Data in Mobile Environments, IEEE Trans. Comput., Vol. 51 No. 10, 2002, 1141–1153