A

Spatial Alarm Processing and Algorithms

Myungcheol Doo, Georgia Institute of Technology
Ling Liu, Georgia Institute of Technology

Spatial alarms are fundamental capability for location based advertisements and location based reminders. One of the most challenging problems in scaling spatial alarm processing is to compute alarm free regions (AFR) such that mobile objects traveling within an AFR can safely hibernate the alarm evaluation process until approaching the nearest alarm of interest. In this paper we argue that maintaining an index of both spatial alarms and empty regions (AFR in the context of spatial alarm processing) is critical for scalable processing of spatial alarms. Unfortunately, conventional spatial indexing methods, such as R-tree family, k-d tree, Quadtree, and Grid, are not well suited to index empty regions. We present Mondrian Tree $-$ a region partitioning tree for indexing both spatial alarms and alarm free regions. We first introduce the Mondrian tree indexing algorithms, including index construction, search, and maintenance. Then we describe a suite of Mondrian tree optimizations to further enhance the performance of spatial alarm processing. Our experimental evaluation shows that the Mondrian tree index outperforms traditional index methods, such as R-tree, Grid, Quadtree, and k-d tree, for spatial alarm processing.

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Spatial Databases and GIS

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Location-based Systems, Data structures, spatial databases

1. INTRODUCTION

A spatial alarm is defined by three elements: *a future reference location* known as the alarm monitoring region and represented typically by a spatial region of interest, *an owner* who is the publisher of the alarm, and *the list of subscribers* of the alarm. In contrast to time-based alarms that remind us of the arrival of a *future reference time point*, spatial alarms remind us of the arrival of a *future reference location*.

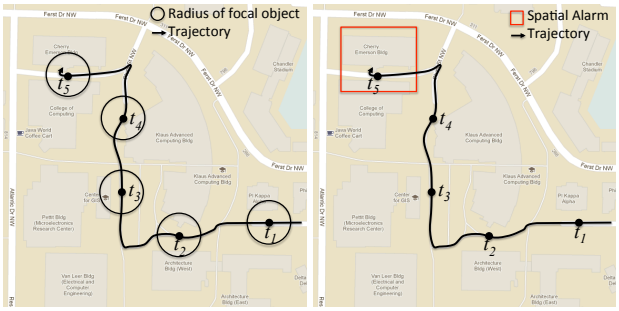
We consider three categories of alarms: *private*, *shared* and *public*. *Private* alarms are installed and used exclusively by the publisher. *Shared* alarms are installed by the publisher with a list of authorized subscribers and the publisher is typically one of the subscribers. *Public* alarms are usually installed with the purpose of sharing them with all mobile users. Public alarms can be useful means of informing subscribers about hazardous road situations, sever weather forecast, or delivering targeted advertisement.

Example 1: Location-based advertisements. North Face is an example of such services and it identifies the handsets of opted-in consumers within a certain radius of retail stores and sends text messages with discount offers [Grove 2010]. This is an example of public alarms with subscriptions.

Example 2: Location-based reminders [Sohn et al. 2005]. Alice sets a spatial alarm on a vitamin store in Lenox Square to *"remind her to pick up some vitamin products when she is within three miles of the store"*. This is an example of a private spatial alarm.

Spatial alarms are fundamentally different from spatial continuous queries. Figure 1 shows how spatial alarms are different from spatial continuous queries. Spatial continuous queries are targeted at mobile objects and events occur in the vicinity of the current location of the mobile user who issued the queries (i.e., the query focal object, represented as circles in Figure 1(a)). An example of spatial continuous query is "find a nearest restroom within 500 meters from the current location." As the user moves on the road, the spatial queries are always centered at the vicinity of the query focal object as shown in Figure 1(a). For example, we search spatial objects within 500 meters from the current location at t_1, t_2, t_3, t_4, and t_5. In contrast, spatial alarms are

targeted at a future reference location of interest, instead of the current location of the mobile subscriber. Consider a spatial alarm such as "remind me to submit a petition for graduation near Cherry Emerson Building." When a user is moving on the road, the continuous query approach keeps looking at the vicinity of the current location to see if the focal object is overlapping with the boundary of Cherry Emerson Building. However, the spatial alarm approach monitors the vicinity of the building if Alice is located near the building.



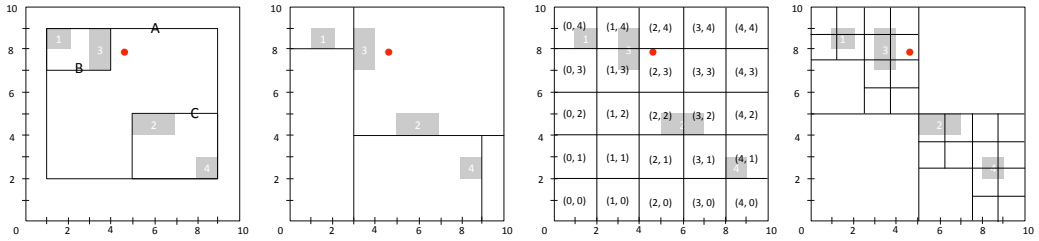
(a) Spatial Continuous Queries (b) Spatial Alarms

Fig. 1. Comparisons between spatial Continuous Queries and spatial alarms

Clearly, if the user is far from the alarm monitoring region, all alarm evaluations are unnecessary and will not result in alarm notification. An ideal approach is to hibernate the spatial alarm application when the user is traveling locally and only trigger it when the user is approaching the Cherry Emerson Building within 200 meters. Thus, an important challenge in scaling spatial alarm processing is to compute alarm free regions such that mobile objects traveling within a rectangular region containing no spatial alarms can safely hibernate the alarm evaluation until approaching the nearest alarms of interest. For example, when user is at t_1 the mobile device starts to hibernating and wakes up at t_4 or t_5. We argue that in the context of spatial alarm processing, spatial alarms and alarm free regions are equally important and both should be treated as the first class citizen.

It is well known that maintaining an index of spatial data of interest is critical for retrieving those spatial data quickly. However, conventional spatial indexing methods, such as R-tree [Guttman 1984], k-d tree [Bentley 1975], Grid [Nievergelt et al. 1984], and Quadtree [Finkel and Bentley 1974], are not well suited to index empty regions. Figure 2 shows how conventional indexing methods index spatial data of interests 1, 2, 3, and 4 (spatial alarms in this context) painted as gray rectangles. The users current location is depicted as a red dot. Note that all structures except Quadtree and Grid do not index empty regions (the non-gray regions.) Given the current user location, R-tree and k-d tree know that the user is not inside of spatial alarms but do know what the size of empty region where user can move around without worry of entering spatial alarms. Grid knows user is in the cell (2,3) and the cell is empty. However, if a user moves to the cell with a spatial alarm such as (1, 3), then Grid cannot tell what the size of empty region without computing the empty region. Like Grid, Quadtree sometimes knows the size of empty region. Given the current location, however, Quadtree cannot tell the size of empty region without AFR computation.

It is not impossible to index empty regions for these conventional index structures. For R-tree and k-d tree we can manually create small empty regions and add them into the index. Then it becomes the Mondrian tree. For Grid the cell size should be smaller



(a) R-tree (b) k-d tree (c) Grid (d) Quadtree

Fig. 2. Indexing spatial alarms using conventional spatial indexing methods

enough so that one of the cell boundaries coincides the borders of spatial alarms. This smaller cell size causes two problems: (a) bigger storage and (b) frequent crossing cells. We can further partition the cell of Quadtree so that the cell boundaries touch alarm boundaries, we immediately know where the empty regions are. Then like Grid, we face same issues. This example illustrates that if we want to index both spatial alarms and the empty regions, the conventional indexes are no longer suitable since those indexes by design work effectively only when some sub-regions in the universe of discourse containing the spatial objects of interest need to be indexed.

In this paper we present the design of Mondrian tree, a region partition index for both spatial alarms and alarm free regions (AFR). We first describe the Mondrian tree index construction, search, and maintenance algorithms. Then we describe a suite of optimizations to further enhance the performance of Mondrian indexing and spatial alarm processing. Mondrian tree index has two unique features. First, it utilizes the pre-computation and indexing of empty regions to avoid on-the-fly computation of alarm free regions based on the motion behavior of mobile subscribers. Second, it incorporates a suite of locality-aware and motion-aware optimizations to further minimize the amount of wakeups and the number of region-crossing checks to be performed at mobile clients. We conduct a set of extensive experimental evaluations and show that the Mondrian tree indexing offers fast spatial alarm processing, and it significantly outperforms existing spatial indexing methods, such as R-tree, k-d tree, and Quadtree, which compute alarm free regions dynamically based on the motion behavior of mobile users.

2. OVERVIEW AND RELATED WORK

A critical challenge for efficient processing of spatial alarms is to determine *when to evaluate* each spatial alarm, while ensuring the two demanding objectives: *high accuracy*, which ensures zero or very low miss rate of spatial alarms, and *high efficiency*, which requires highly efficient processing of spatial alarms.

Periodic evaluation can be performed for spatial alarms by checking whether a mobile subscriber is entering the spatial alarm on every pre-defined time interval. High frequency is essential to ensure that none of the alarms are missed. Though periodic evaluation is simple, it can be extremely inefficient due to frequent alarm evaluation and the high rate of irrelevant evaluations [Murugappan and Liu 2008].

Similarly, **processing spatial alarms upon location updates** of mobile users is equally incompetent and wasteful due to the specific characteristics of spatial alarms. For example, assume that the user is currently at t_1, 10 miles away from her spatial alarm as shown in Figure 1(b). Then it is unnecessary to evaluate those spatial alarms upon her location updates when she approaches at t_4 or t_5 [Bamba et al. 2008].

Safe regions are popular techniques for continuous spatial query processing [Gedik and Liu 2004; Hu et al. 2005; Prabhakar et al. 2002; Hasan et al. 2009; Hu et al. 2005]. The safe region of an object o is dynamically computed at the server based on the set of queries such that the current results of all queries remain valid as long as o is residing inside its safe region. Computing safe region takes $O(n^2)$ for n queries [Prabhakar et al. 2002]. Although [Bamba et al. 2008] extends the safe regions to spatial alarm processing, the high cost of dynamic safe region computation remains to be a challenging problem. As the mobile client moves, the server needs to re-compute its new safe region continuously, which can be expensive. Figure 3 shows spatial alarms installed near the Lombard street, San Francisco. At t_0, the safe region is r_1, the wide rectangle, because it is the largest rectangle that does not overlap with any spatial alarms. At t_1, the client exists r_1 and the server computes a new safe region, r_2, the tall rectangle. At t_2, however, the client gets out of r_2 and the server computes a new safe region which is the same as r_1. The computation of the same safe region, r_1, occurs at t_0, t_2 and t_4 as the client moves along the Lombard street. This example scenario shows that although the safe region approach reduces the amount of unnecessary alarm processing, it also introduces a fair amount of unnecessary safe region computation.

Bearing these issues in mind, we present the Mondrian[1] tree indexing structure, which partitions the universe of discourse into smaller regions of two types: spatial alarm regions and empty regions. We call empty regions Alarm Free Regions (AFR) because there is no alarm inside of the region. With Mondrian indexing, it takes $O(\log n)$ for searching the AFR of the mobile client, which is much more efficient compared to $O(n^2)$ for computing safe region on the fly. To our best knowledge, Mondrian tree is the first index structure to partition the universe of discourse into small regions containing objects of interest (spatial alarms in our context) and empty regions and index them all.

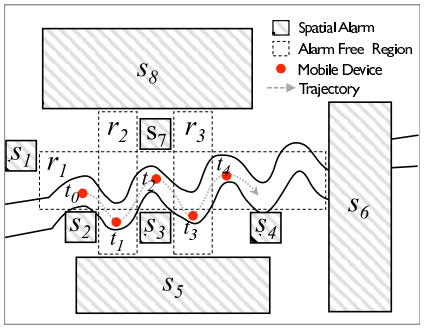


Fig. 3. Safe regions computed at time t_i

3. BASIC MONDRIAN TREE INDEX

In this section we introduce the basic Mondrian tree index, including data structure, batch index construction, search, and insertion algorithm. Although Mondrian tree index is a general indexing structure and can be used to index any type of spatial objects, in this paper we will introduce it in the context of spatial alarms.

3.1. Definitions and Notations

Given a universe of discourse containing a set of m spatial alarms $\{o_j | 1 \leq j \leq m\}$, the corresponding Mondrian tree has a set of n nodes: $\{v_i | 1 \leq i \leq n\}$. Each v_i stores the partitioned disjoint rectangular region, which is either an empty region or a region containing o_j. Without loss of generality, we denote each spatial alarm o_j by its minimal bounding rectangle MBR_j. The leaf nodes of a Mondrian tree are either MBR (spatial alarms) or AFR (empty regions). The internal nodes of the Mondrian tree contain a set of pointers to its children nodes. We represent and store each spatial region in Mondrian tree as points using a point transformation scheme [Seeger and Kriegel 1988]. A k-dimensional rectangle shaped region with each side parallel to one dimensional axis

[1]The name Mondrian is named after Piet Mondrian because region partitioning resembles his "Composition with red, yellow blue and black".

is represented by a $2k$-dimensional point. For example, a 2-dimensional rectangular region r is defined by its lower left corner and upper right corner as a 4-dimensional point r:

$$r = (p_0, p_1, p_2, p_3),$$

where (p_0, p_1) is the lower left corner and (p_2, p_3) is the upper right corner of the rectangle. From now on, we use $r[i]$ to denote the $(i+1)$-th coordinate value of a 4-dimensional point r. For example, $r[2]$ is p_2. If r is an array, then $r[i]$ is the $(i+1)$-th item. Each node v_i consists of 5 components:

$$v_i = (R_i, \text{OID}_i, \text{SI}_i, K_i, \text{CHILD}_i).$$

R_i is a disjoint rectangular region that v_i represents and expressed as a $2k$-dimensional point as explained above. OID_i is a set of identifiers of the spatial alarms, which overlaps with the rectangular region R_i. Each non-leaf node is split into two children nodes. SI_i, a split index, and K_i, a key, are used for performing node split of internal node v_i. SI_i decides which axis to use for partitioning the node v_i and K_i is the value of the split line. SI_i is an integer value between 0 and $2k-1$ and is defined as:

$$\text{SI}_i = d_i \; mod \; 2k,$$

where d_i is depth of v_i. If SI_i is zero or an even integer value, then we split the internal node v_i along the line parallel to x axis, otherwise, along the line parallel to y axis. When splitting on v_i containing o_j, K_i is $\text{MBR}_j[\text{SI}_i]$. CHILD_i is a set of pointers to either null or the children nodes of v_i. Leaf nodes do not have K_i because they are either empty regions or MBR_j and thus nonsplit nodes. We set the split index SI_i of leaf nodes to be -1. For the root node, v_0, its depth d_0 is 0. Given k as 2, SI_0 is 0, which is computed by (0 mod 2).

3.2. Region Partitioning through Node Split

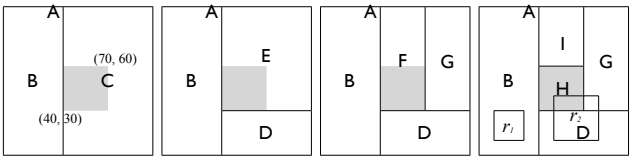
Given a node v_i, partitioning a node is processed using SI_i, K_i, and o_j. For each o_j, we have $\text{MBR}_j = (p_0, p_1, \ldots, p_{2k-1})$. Assuming that the value of the split index SI_i is set to 0. Then $K_i = \text{MBR}_i[\text{SI}_i]$ and $\text{MBR}_i[\text{SI}_i] = \text{MBR}_i[0] = p_0$, and we split the node v_i along the line parallel to the y axis represented by $x = p_0$. Now the region of v_i is split into two smaller disjoint regions, denoted by $\text{CHILD}_i[0]$ and $\text{CHILD}_i[1]$. One of the two children regions contains o_j. Assuming that it is $\text{CHILD}_i[1]$, then the node split process for v_i repeats in the region of $\text{CHILD}_i[1]$. In each iteration SI_i and K_i are computed again, and the node split of v_i is performed along the line parallel to the x or y axis determined by SI_i and K_i. This node split process iterates until every p_l $(0 \leq l \leq 2k-1)$ in R_i is examined.

Figure 4 shows how to partition the region. Figure 5 shows the Mondrian tree with respect to Figure 4 and Table I shows how data is stored in the Mondrian tree. Node v_0 is the root node that covers the rectangle region A represented by $(0, 0, 100, 100)$. Initially we set SI_0 as 0. MBR_0 for the spatial alarm o_0 is $(40, 30, 70, 60)$. In step 1, we compute the key K_0 by $\text{MBR}_0[\text{SI}_0]$, which is 40. Then v_0 is divided into two children along the line parallel to y-axis represented as $x = 40$. Now v_0 has two children v_1 for rectangle B and v_2 for rectangle C as shown in Figure 4(a). R_1 for v_1 is $(0, 0, 40, 100)$, which is computed from $(0,\ 0,\ K_0,\ 100)$ and R_2 for v_2 is $(40,\ 0,\ 100,\ 100)$, computed from $(K_0, 0, 100, 100)$. In step 2, v_2, rectangle C, is chosen to be partitioned because o_0 intersects with v_2. v_2 is divided into v_3, rectangle D, and v_4, rectangle E as shown in Figure 4(b). In step 3, v_4 is chosen and partitioned into v_5, rectangle F, and v_6, rectangle G. At last, v_5, rectangle F, is divided into v_7, rectangle H, and v_8, rectangle I. As a result, this example Mondrian tree indexes both the spatial alarm represented by

Table I. Data stored in the Mondrian tree

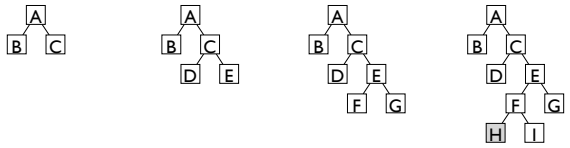
v_i	MBR$_i$	SI$_i$	K_i
v_0 (A)	$(0, 0, 100, 100)$	0	40
v_1 (B)	$(0, 0, 40, 100)$	1	N/A
v_2 (C)	$(40, 0, 100, 100)$	1	30
v_3 (D)	$(40, 0, 100, 30)$	2	N/A
v_4 (E)	$(40, 30, 100, 100)$	2	70
v_5 (F)	$(40, 30, 70, 100)$	3	60
v_6 (G)	$(70, 30, 100, 100)$	3	N/A
v_7 (H)	$(40, 30, 70, 60)$	0	N/A
v_8 (I)	$(40, 60, 70, 100)$	0	N/A

a gray rectangle H using index node v_7 in Figure 4 and Figure 5 but also four empty regions denoted by B (v_1), D (v_3), G (v_6), and I (v_8).



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 4. Partitioning Nodes



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 5. Mondrian tree for Figure 4

3.3. Search Operation

In spatial alarm processing, we use the Mondrian tree to find whether a mobile user enters a spatial alarm region or stays in an alarm free region which is an empty region. We refer to this search operation as a point search. In addition, we also need a region search operation that can find which index node overlaps with a given rectangle region.

Point search over the Mondrian tree is to find a leaf node node v_i that contains a point p. For example, in the context of two dimensional data space, the point search can be used to answer a query like "What is the smallest rectangular region that contains the point (x, y)?" or "Does the mobile user at position (x, y) enter a spatial alarm region?".

Given a point p in k dimensional space, POINTSEARCH in Algorithm 1 is the point search algorithm that takes the Mondrian tree and the point p as input and outputs the leaf node (either a spatial alarm region or an empty region) in which p resides. The algorithm starts the search from the root node. For each node v_i, POINTSEARCH determines which dimension is used for v_i to split the node (line 4). For this dimension, we compare coordinate of p with the key, K_i, associated with v_i. If $p[Key]$ is greater

ALGORITHM 1: POINTSEARCH

Input: Node v_i and current location p.
Output: A leaf node v that contains p.

```
1 if v_i = leaf then
2     return v_i
3 end
4 Dim ← SI_i mod k                  // Determine split line dimension; 0 for x or 1 for y
5 if p[Dim] < K_i then
6     return POINTSEARCH(CHILD_i[0], p)          // p is in left or bottom of split line
7 else
8     return POINTSEARCH(CHILD_i[1], p)           // p is in right or top of split line
9 end
```

than or equal to K_i, then POINTSEARCH searches the right subtree because p is located on right side (or upper side) of the split axis. Otherwise, the left subtree will be searched. POINTSEARCH repeats this process iteratively until it reaches the leaf node that contains p.

The time complexity of the point search operation on a Mondrian tree is proportional to the height of the tree given its binary search feature. The average height of a binary search tree with n nodes is $\log n$. Therefore, the time complexity of POINTSEARCH is also $O(\log n)$. For the skewed Mondrian tree, the height might be n. Then the worst time complexity of POINTSEARCH is $O(n)$.

Region search algorithm REGIONSEARCH finds all leaf nodes in the Mondrian tree, which overlap with a given rectangle r. Concretely, REGIONSEARCH first examines the root node v_i to find if one of its two children nodes overlaps with r. Let c be $\text{SI}_i \ mod \ k$ and c be $(c + k) \ mod \ k$. Unlike POINTSEARCH, REGIONSEARCH compares v_i's key with an interval. The interval is given by an upper bound and a lower bound in the c-th dimension of the given rectangle r, denoted by $\text{Min}(r[c], r[c])$ and $\text{Max}(r[c], r[c])$. If K_i is greater than or equal to the upper bound, then the left children $\text{CHILD}_i[0]$ will be taken as the input and the algorithm REGIONSEARCH starts the new iteration with $\text{CHILD}_i[0]$ and r. If K_i is smaller than or equal to the lower bound, then the right children $\text{CHILD}_i[1]$ will be taken as the input and the algorithm REGIONSEARCH starts the new iteration with $\text{CHILD}_i[1]$ and r. Otherwise, both children of v_i overlap with r. Thus, both REGIONSEARCH(CHILD_i[0], r) and REGIONSEARCH(CHILD_i[1], r) are invoked. This process repeats until the leaf nodes are reached.

Consider the Mondrian tree in Figure 5(d) and a rectangle $r_1(10, 10, 20, 30)$ as shown in Figure 4(d), the algorithm REGIONSEARCH returns v_1 representing region B. Given a rectangle r_2(50, 10, 70, 40), the algorithm REGIONSEARCH returns three leaf nodes, v_3, v_6, and v_7, which represent rectangle regions D, G, and H, all overlapping with r_2. Concretely, r_1's x interval is (10, 20) and y interval is (10, 30). Given that K_0 as 40, x's upper bound, 20, is smaller than K_0. Therefore the rectangle r_1 is overlapping with $\text{CHILD}_0[0]$, which is the rectangle region B and located at left of the split line. Because B is the leaf node, REGIONSEARCH finishes returning B. Similarly, for r_2 x interval is (45,65) and y interval is (10,40). Thus, K_0 is smaller than the x's lower bound of r_2, which is 45. That is r_2 is overlapping with v_2. We compare y interval with K_2 and K_2 is staying between y interval. Therefore REGIONSEARCH launches two REGIONSEARCH, one with v_3 and the other with v_4. The first launch finishes because v_4 is the leaf node. The second launch also launches two REGIONSEARCH. We keep going down until all REGIONSEARCH reach leaf nodes. As a result we have D, G, and H.

ALGORITHM 2: REGIONSEARCH

Input: Node v_i and rectangular region r.
Output: A set of leaf nodes that overlaps with r.

```
1  if v_i = leaf then
2      return v_i
3  end
4  c ← SI_i mod k                                  // Determine index of one end of range
5  c' ← (c + k) mod k                              // Determine index of the other end of range
6  min ← Min(r[c], r[c'])                          // Left end of range
7  max ← Max(r[c], r[c'])                          // Right end of range
8  if max ≤ K_i then
9      return RegionSearch(Child_i[0], r)          // r is in left of split line
10 end
11 else if K_i ≤ min then
12     return RegionSearch(Child_i[1], r)          // r is in right of split line
13 end
14 // r is overlapping with split line
15 nodes_left ← RegionSearch(Child_i[0], r)
16 node_right ← RegionSearch(Child_i[1], r)
17 return nodes_left ∪ nodes_right
```

Similar to a point query, REGIONSEARCH finishes if it arrives at the leaf node. Therefore, REGIONSEARCH also takes $O(\log n)$ on average and $O(n)$ for the worst case.

3.4. Insertion

The algorithm of inserting a new spatial alarm o_j to a Mondrian tree starts from the root node and find a node v_t using REGIONSEARCH. v_t is either a non-leaf node whose children both overlap with MBR_j or a leaf node. Then we consider three cases:

(i) $\text{MBR}_j \cap R_t = \text{MBR}_j$. MBR_j is smaller than and fully contained in R_t of a leaf node v_t. Then we split v_t into $\text{CHILD}_t[0]$ and $\text{CHILD}_t[1]$, such that only one of them fully contains MBR_j, say $\text{CHILD}_t[1]$. Now INSERT is invoked with $\text{CHILD}_t[1]$ and o_j (Lines 5-10).

(ii) $\text{MBR}_j = R_t$. MBR_j is the same as R_t. Then we add the identifier of oj into OID_t and stop the algorithm (Line 3).

(iii) Otherwise, v_t is a non-leaf node and MBR_j is overlapping with both children nodes of v_t. Then we split MBR_j into two disjoint partitions along the line computed by SI_t and K_t. Now the insertion algorithm is invoked to insert two partitions of o_j into the two children nodes of v_t one at a time (Lines 13-15).

We provide a sketch of the pseudo code in Algorithm INSERT. Given a Mondrian tree of n nodes, the time complexity of INSERT is $O(\log n)$. INSERT consists of two parts: (1) find v_t and (2) partition v_t or MBR_j. First part takes usually $O(\log n)$. The second part takes $O(1)$.

Although Mondrian tree is a memory-based tree, it can be extended to a tree with page-oriented storage like hard disks. Concretely, if the capacity of memory is n nodes, then those n nodes are stored in the memory and the remaining nodes are stored on external pages. We adopt this structure from LSD tree [Henrich et al. 1989]. In this paper, we briefly describe basic ideas of using external pages. For more detailed algorithm, you can refer the LSD paper. Figure 6 shows the combination of memory and external storage. If the size of M exceeds n, then the subtree of M is written into an external page. If the height of a subtree in external pages exceeds h_p, which is set by the system, then the external page is split into two pages.

ALGORITHM 3: INSERT

Input: Node v_i and spatial alarm o_j.

```
1  if v_i is leaf then
2      if MBR_j = R_i or OID_i ≠ null then
3          Add O_j.ID into OID_i                                  // case ii
4      else
5          splitNode(v_i)                                         // case i
6          if CHILD_i[0].R contains MBR_j then
7              INSERT(CHILD_i[0], o_j)
8          else
9              INSERT(CHILD_i[1], o_j)
10         end
11     end
12 else
13     (o_j.left, o_j.right) = splitAlarm(o_j)                    // case iii
14     INSERT(CHILD_i[0], o_j.left)
15     INSERT(CHILD_i[1], o_j.right)
16 end
```

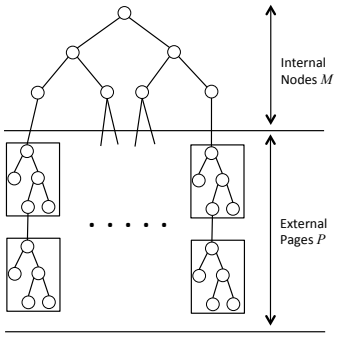


Fig. 6. Memory and external storage

3.5. Deletion

The deletion algorithm consists of two steps. The first step is to find nodes that have the alarm to be deleted. Given the alarm, a set of nodes that overlaps with the alarm can be found if we maintain a hash table for a pair of alarm identification and a set of nodes that contains the alarm. Then for each found node v, we remove the alarm from v. The second step handles the merge operation. For example, in Figure 5(d), if we remove the alarm from H, then we can remove all nodes except A. As a result, this delete operation is the same as merge operation.

ALGORITHM 4: DELETE

Input: Set of nodes that contains o_j

```
1 foreach Node v in set of nodes do
2     v.removeID(o_j.ID)
3     MERGE(v)
4 end
```

ALGORITHM 5: MERGE

Input: Node v

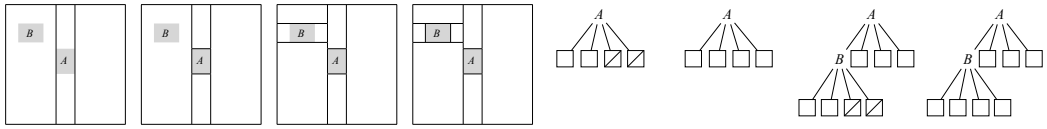
```
1 p ← v.getParent()
2 if p ≠ NULL and p's children has no alarms then
3     p.removeChildren()
4     MERGE(p)
5 end
```

4. MONDRIAN*: AN OPTIMIZED TREE

The basic Mondrian index does not guarantee the balance of the tree. Once a node is partitioned into two disjoint smaller regions, one of them is not touched and the other is selected for further partition. Therefore the basic Mondrian index cannot guarantee the balance of the tree. Also the skewness of the tree is affected by the insertion order. In this section we introduce Mondrian*, an optimized Mondrian index structure that produces a more balance tree.

The Mondrian* tree also has similar **data structure** except that each node has four children instead of two, denoted by $\text{CHILD}_i[0]$, $\text{CHILD}_i[1]$, $\text{CHILD}_i[2]$, and $\text{CHILD}_i[3]$. Each rectangular region is stored at each node using four pointers instead two in the basic Mondrian tree. Given v_i, CHILD_i has pointers to left, right, bottom and top of MBR_i. Thus, the Mondrian* tree has four keys in each node because we split the space by four lines.

Figure 7 shows an example of how the Mondrian* partitions the space. For the chosen spatial alarm, say A, the Mondrian* partitions the space by drawing two vertical lines along the alarm as shown in Figure 7(a). Now we set $\text{CHILD}_0[0]$ and $\text{CHILD}_0[1]$ as shown in Figure 7(e). A box with a diagonal line means that node is null. Then two horizontal lines are drawn as shown in Figures 7(b) and we set $\text{CHILD}_0[2]$ and $\text{CHILD}_0[3]$ as shown in Figure 7(f). When B is inserted, then we put B on $\text{CHILD}_0[0]$ because B lies on left of A. Then we perform the same region partition operation on B as shown in Step 3 and Step 4 of Figure 7.



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4 (e) Step 1 (f) Step 2 (g) Step 3 (h) Step 4

Fig. 7. Region Partition in Mondrian*

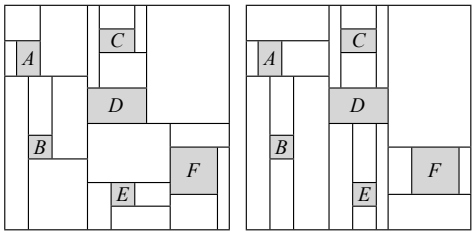
Mondrian* Index Batch Construction. Given a set of n spatial alarms, the worst case in index construction occurs when every spatial alarm $o_j (1 \leq j \leq m)$ is inserted as a child at the lowest leaf node. One advantage of batch index construction is to utilize the prior knowledge on the distribution of spatial alarms to avoid the extreme skewedness of the tree. In this section we describe the batch construction method of Mondrian* index such that no subtree has more than one half of the nodes in the Mondrian* tree. For presentation brevity, we use k as 2 for two dimensional space as our context.

The first step is to sort the set of n spatial alarm objects in x coordinate and secondly in y coordinate. Given a sorted list of spatial alarm objects, we choose the spatial alarm object that is the medium of the ordered list as the root node of the Mondrian* tree, denoted by o_{root}. We insert this o_{root} first by performing the region par-

tition as discussed above (recall example in Figure 4). The four children of o_{root} are created and denoted as $\text{CHILD}_{root}[0]$ (left), $\text{CHILD}_{root}[1]$ (right), $\text{CHILD}_{root}[2]$ (bottom), and $\text{CHILD}_{root}[3]$ (top). Furthermore, the remaining spatial alarm objects in the ordered list are regrouped into four sub-collections, each of which will be inserted into one of four children of the root. Those spatial alarms that are ranked before the chosen root object will be placed with $\text{CHILD}_{root}[0]$ (left) or $\text{CHILD}_{root}[2]$ (bottom) and the remaining spatial alarms that are ranked after the chosen root object will be indexed through $\text{CHILD}_{root}[1]$ or $\text{CHILD}_{root}[3]$. This process iterates recursively until all alarm objects in the ordered list are inserted in the Mondrian* tree. Clearly, this batch construction process ensures that no subtree can possibly contain more than half of the total number of nodes.

The time complexity of this algorithm is $O(n^2 \log n)$, given the ordering step on n spatial alarm objects takes $O(n \log n)$, the selection of the median requires $O(1)$, and INSERT takes $O(n)$.

Figure 8 shows the result of region partitioning by basic Mondrian tree and Mondrian* tree after the batch index construction over six spatial alarms. Figure 9 shows the corresponding Mondrian and Mondrian* tree. Although the spatial alarm D is chosen as the root node for both basic Mondrian and Mondrian*, the region partitioning result for Mondrian* is quite different than basic Mondrian tree (see Figure 8). So is the index structure (as shown in Figure 9). This example illustrates that Mondrian* is much more balanced compared to the basic Mondrian index and thus offers much higher efficiency in terms of search and insertion.



(a) Mondrian (b) Mondrian*

Fig. 8. Region Partitioning: An Example

Search algorithm for Mondiran* works in a similar way as the basic Mondrian search. The only change that we need to add to the Mondrian* search is the index key comparison part since each node in Mondrian* has four keys, $K_i[0]$, $K_i[1]$, $K_i[2]$, and $K_i[3]$. Given a point p and a node v_i in Mondrian*, the point search algorithm uses $\text{DIRECTION}(p, v_i)$ to compute and return a pointer to one of v_i's four children, which contains p, by comparing p's coordinate value with K_i. For example, if $\text{DIRECTION}(p, v_i)$ returns 2 then we visit $\text{CHILD}_i[2]$.

5. SPATIAL ALARM EVALUATION

We have presented Mondrian tree and Mondrian* tree for indexing spatial alarms and empty regions. Intuitively, we can treat each empty region as an alarm free region (AFR) such that when a mobile user travels inside an AFR, the alarm evaluation is hibernated, saving both energy consumption at the mobile client and the alarm evaluation cost at the server. However, as shown in Figure 8, an empty region in Mondrian tree may not be the best AFR for a mobile user, especially when multiple AFRs are adjacent to one another. In this section we describe the best strategies for evaluating

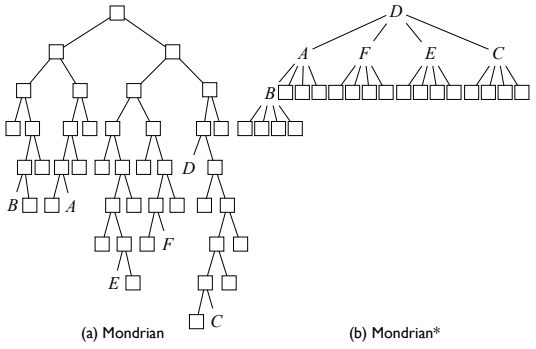


Fig. 9. Mondrian v.s. Mondrian*: an example

spatial alarms using Mondrian tree indexes, including alarm free period, patched and trimmed AFR, motion-aware AFR and distributed Mondrian indexing scheme.

In principle, a spatial alarm should be evaluated in three steps. First, we need to determine what type of events should activate the spatial alarm evaluation process. Second, the server needs to find out the list of alarms to be evaluated upon the occurrence of the alarm events. The shorter this list is, the more efficient the spatial alarm evaluation will be. Third, the server executes the action component of those spatial alarms whose alarm conditions are evaluated to be true.

As discussed in Section 2, periodic evaluation is extremely inefficient due to frequent alarm evaluation and high rate of irrelevant evaluations. Although using the location update of a mobile user as the alarm evaluation event seems appealing, and it is independent of the concrete location update strategies, such as periodic, dead-reckoning or others [Pesti et al. 2010], we have pointed out in Section 2 that many location update events are not suitable as the alarm evaluation events. First, not all location updates of a mobile user will lead to a successful evaluation of her spatial alarms, especially when she travels in the spatial area that does not contain any of her spatial alarms. Second, location updates of a mobile user will have zero probability of leading to successful evaluation of those spatial alarms that are not owned or subscribed by this mobile user. For instance, Bob's private spatial alarms are indifferent to the location updates of Alice.

To address the first issue, we promote the use of alarm free regions such that no spatial alarm evaluation will be activated when a mobile user travels inside an alarm free region. This can significantly reduce the frequency and overhead of spatial alarm evaluation. To address the second issue, the server needs to find out the list of alarms to be evaluated upon the occurrence of the alarm events. The shorter is this list, the more efficient is the spatial alarm evaluation. This motivates us to design the distributed Mondrian tree index structure.

5.1. Alarm Free Period

An obvious idea for evaluating spatial alarms efficiently is to incorporate the spatial locality of the alarms and the motion behavior of mobile objects through alarm free regions. We have discussed in Section 2 that dynamic computation of alarm free regions is expensive due to unnecessary and possibly duplicate AFR computation. Given that Mondrian tree approach indexes both spatial alarm regions and empty regions, it is intuitive to use empty regions as alarm free regions (AFRs). The only cost for using AFRs is time to lookup the leaf node instead of computing AFRs. Once AFR is acquired,

then the client needs to check if it is still inside of AFR. We introduce the concept of alarm free period (AFP) as a basic approach to assist a mobile user to determine when to check whether she moves outside of her current AFR. An important property of AFP is that it should avoid missing alarms or minimize the alarm miss rate.

5.1.1. Basic AFP. Given a mobile client m and an alarm free region AFR_m, the AFP_m is the shortest travel time for m to arrive at the closest border of its current alarm free region AFR_m. During AFP_m, m's alarm evaluation service can enter a sleep (hibernate) mode.

Two main factors that impact on the computation of the AFP_m are the velocity of m, say V_m, and the shortest distance from the current position of m to the closest border of AFR_m, say $MinDist(m, \text{AFR}_m)$. Thus, the AFP_m can be computed as follows:

$$\text{AFP}_m = \frac{MinDist(m, \text{AFR}_m)}{V_m} \tag{1}$$

One caveat with Equation 1 is that it assumes that the mobile subscriber m moves in a straight line from her current location to the closed border of AFR_m. It is, however, not a realistic assumption in real life. For example, it is highly likely that the mobile user m is moving towards a direction that is opposite of the closest border of its current AFR_m. The steady motion assumption is specifically true when mobile users move on the road networks.

5.1.2. Steady Motion based AFP. Given a mobile user m and m's previous moving direction θ, we can compute the moving direction of m using the probability density function of moving direction θ, denoted by $p(\theta)$. $p(\theta)$ is uniformly distributed and $p(\theta)$ is $\frac{1}{2\pi}$ if the mobile client selects the next direction randomly, as shown in Figure 10(a).

However, under the steady motion assumption, the mobile client is likely to increase or decrease the value of θ but not dramatically. For example, at an intersection, the probability of making a U-turn for the mobile client is less than the probability of making a left or right turn. Therefore, the density function $p(\theta)$ is not uniformly distributed. The modified $p(\theta)$ is provided as follows:

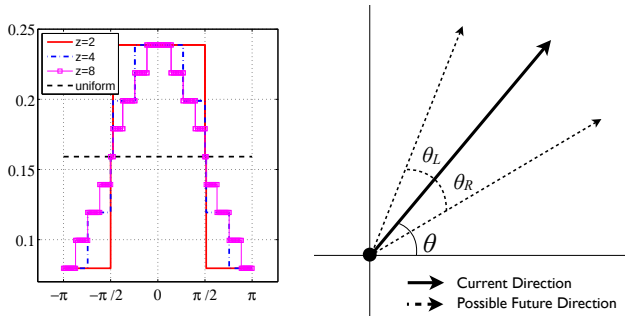
$$p(\theta) = \begin{cases} \frac{1+\frac{y}{z}\lceil\frac{\frac{\pi}{2}-|\theta|}{\frac{y\pi}{z}}\rceil}{2\pi} & \textit{if } \theta \in [-\frac{1}{2\pi}, \frac{1}{2\pi}] \\ \frac{1-\frac{y}{z}\lceil\frac{-\frac{\pi}{2}+|\theta|}{\frac{y\pi}{z}}\rceil}{2\pi} & otherwise \end{cases}$$

Here y and z are parameters of steadiness such that $\frac{y}{z} < 1$. Figure 10(a) shows the probability density function $p(\theta)$ for different values of z when $y = 1$.

Figure 10(b) shows the steady motion behavior over the moving direction θ while θ may change between $-\theta_L$ and $-\theta_R$. Based on this assumption we define a steady motion distance $D_{steady}(m, \text{AFR}_m)$, as follows:

$$\frac{1}{\theta_R + \theta_L} \int_{-\theta_L}^{+\theta_R} D_\theta(m, \text{AFR}_m) d\theta \tag{2}$$

where $D_\theta(m, \text{AFR}_m)$ is the distance from the current location of m to the intersection point with the boundary of AFR_m over which m may cross while heading towards the θ direction.



(a) $p(\theta)$ (b) Moving Direction

Fig. 10. Steady Motion Assumption

The steady motion based alarm free period for mobile client m, denoted by AFP_m, is computed as follows:

$$\text{AFP}_m = \frac{D_{steady}(m, \text{AFR}_m)}{V_m} \tag{3}$$

5.2. Extending Alarm Free Regions

We have shown that the Mondrian tree indexes both spatial alarm regions and empty regions, and thus we can efficiently determine whether a mobile user is inside of a spatial alarm region or an alarm free region. Furthermore, by utilizing alarm free region (AFR), we can significantly reduce the number of unnecessary alarm evaluations in anticipation of mobile client movement.

However, directly using empty regions as alarm free regions can incur higher number of region crossing checks to be performed, especially when mobile clients travel from one small empty region to another. Thus, the gain from reduction of the number of unnecessary alarm evaluations is offset by the cost of higher AFR crossing checks when AFRs are small in size. We below examine the cost of AFR based alarm evaluation in order to better understand the impact of the AFR perimeter on the cost of alarm evaluation.

Assume that the mobile user m moves in a randomly chosen direction with a constant speed V_m, C_{SA} is the cost for one alarm evaluation, θ is the angle between m's moving direction and the positive x-axis (as shown in Figure 10(b)), $l(\theta)$ is the distance from the current location of m to the intersection point p with the boundary of AFR_m when m travels along the θ direction, $\lambda(\text{AFR}_m)$ is the perimeter of AFR_m, and V_m is m's current speed. Given m's current alarm free region AFR_m, we can compute the amortized alarm evaluation cost for m over time, denoted by C_m, as follows:

$$C_m = C_{SA} \times \left(\int_0^{2\pi} \frac{l(\theta)d\theta}{2\pi \cdot V_m}\right)^{-1} = \frac{C_{SA} \cdot 2\pi \cdot V_m}{\lambda(\text{AFR}_m)} \tag{4}$$

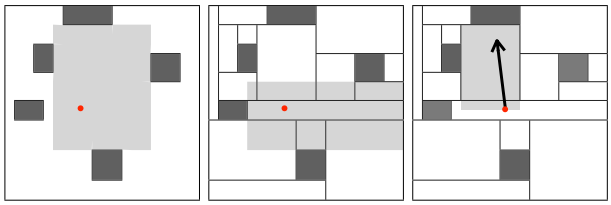
Note that $\int_0^{2\pi} l(\theta)d\theta = \lambda(\text{AFR}_m)$, and given that AFR_m is a rectangle region, the intersection point p should be unique. Based on this equation, the average alarm evaluation cost is minimized when the perimeter of the AFR_m is maximized.

This motivates us to investigate the opportunities of composing larger AFRs for each mobile client by merging empty regions in the vicinity of the mobile client. Intuitively, by maximizing the perimeter of AFRs, we can minimize the number unnecessary region crossing checks to be performed, which further minimize the average cost of alarm evaluation. This is because reducing region crossing checks can lead to better energy

efficiency at mobile clients and reduced communication and computation load at the server.

In the rest of this section, we describe two optimization techniques for extending AFR.

5.2.1. Patch and Trim (PAT). Before describing our technique for merging empty regions to form a larger AFR, we illustrate the technical challenge of this problem by example. Figure 11(a) has five spatial alarms shown in small dark grey rectangles and the red circle denotes the current location of mobile client m. Clearly the optimal AFR with respect to these five spatial alarms and the current location of m is the light gray rectangle in Figure 11(a). However, it is costly in general to compute such an optimal AFR. According to [Chazelle et al. 1984], given a set of n spatial alarms, the time complexity for computing the largest empty rectangle with respect to the n alarms is $O(n \log^3 n)$. Therefore, in this paper we develop a near-optimal but fast algorithm to compute an extended AFR. We call it Patch and Trim (PAT).



(a) Optimal (b) PAT (c) MPAT

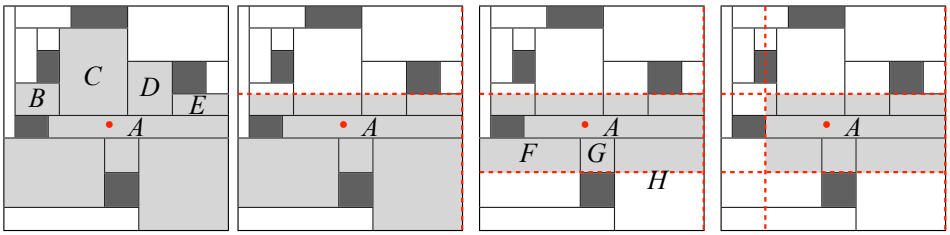
Fig. 11. Examples of extending AFRs

Figure 11(b) shows the result of constructing an AFR rectangle by patching the adjacent AFRs (empty regions) and trimming the patched polygonal region over orthogonal lines.

PAT consists of two phases: $Patch$ phase and $Trim$ phase. In the $Patch$ phase, we search the Mondrian index to get a set of adjacent AFRs with respect to the current location of mobile client m by using REGIONSEARCH in $O(\log n)$. Then we perform the $Trim$ phase over the four sides (top, right, bottom, left) of the patched empty region in the clockwise manner in $O(1)$. In total, PAT takes $O(logn)$, which is faster than $O(n \log^3 n)$.

Figure 12 illustrates the patch phase and the four steps of the $Trim$ phase. Assume that the user is inside of empty region A. Figure 12(a) shows the resulting polygonal area of the patch phase where all neighboring empty regions of A are selected. In the trim phase, we need to trim the polygonal area into a rectangle region containing A by setting the boundary of the extended AFR. This is done by selecting the intersecting interval of neighboring empty regions on four sides of A, clockwise one at a time. For example, in order to extend A upward, we consider A's neighboring empty regions: B, C, D, and E. The intersecting y interval by all four rectangles is the same as E's height. Therefore we choose y value of top border in E as the extended top boundary shown in 12(b). On the right side of A, there are no adjacent AFRs. Therefore we use the x value of A's right border as the right boundary of extended AFR, shown in Figure 12(b). In order to extend A downward, we examine A's downward neighboring empty regions: F, G, and H and the intersecting y interval they share is the same as G's y interval (height). Thus we select y value of G's bottom border as the bottom boundary of the extended AFR, shown in Figure 12(c). Finally we examine if we can extend A on

its left border. Given that the left neighboring region of A is a spatial alarm, we cannot extend A further on its left side. Thus, we choose the x value of A's left border as the left boundary of the extended AFR, shown in Figure 12(d).



(a) Patch (b) Top and Right (c) Down (d) Left

Fig. 12. Patching and Trimming Steps

We have shown that PAT can quickly compute an extended AFR that is near optimal when there is no additional knowledge about user's mobility and motion behavior. As discussed in Section 5.1, given the current AFR of a mobile user m, we compute the alarm free period (AFP) during which m can hibernate its alarm evaluation service. Obviously, the longer is an AFP, the less number of AFR crossing checks will be performed and thus less alarm evaluations for m.

Given that most of mobile users travel on a spatially constrained road network or walk path with a destination in mind, we know the approximate travel direction for these mobile users. Under such steady motion assumption, we can utilize the moving direction of a mobile user to compute the extended AFR such that the alarm free period (AFP) computed using this extended AFR will be maximized.

Recall the example in Figure 11, if user m is moving northwest, then all spatial alarms located in the south of m's current location are no longer relevant. Thus, the best extended AFR for m in this case is the extended AFR computed using MPAT, shown in Figure 11(c). This is because by utilizing the steady motion of m, the extended AFR in Figure 11(c) maximizes the AFP for m.

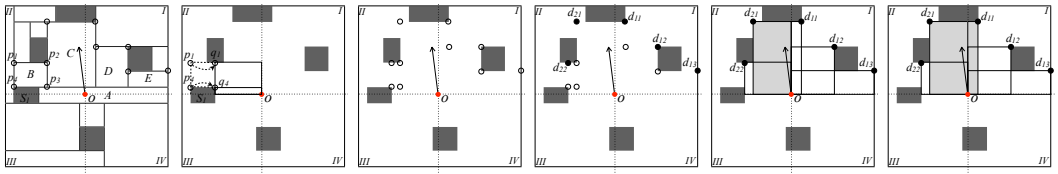
This motivates us to develop a motion-aware patch and trim algorithm that can compute the extended AFR for each mobile user m based on her motion behavior, aiming at maximizing the AFP for m.

5.2.2. Motion-Aware Patch and Trim (MPAT). The motion-aware patch and trim algorithm for extending AFR consists of five steps. Due to space limit, we omit the pseudo code in this paper and provide a walk-through of the algorithm using the example in Figure 13.

Step 1: Determine the relevant Quadrants. Under the steady motion assumption (recall section 5.1), if a mobile user is heading towards the θ direction, then the probability of m moving forward or turning left or right on its current location is much higher than the probability of making a U-turn, as shown in Figure 10(a). Therefore, spatial alarms in the downward direction of m are no longer relevant.

In order to make the best use of the steady motion behavior of the mobile user m, we partition the universe of discourse into four quadrants using the current location of m as the center such that only those quadrants that are relevant to computing the extended AFR of m will be selected.

Figure 13(a) shows an example of four quadrants partitioned at o, the current location of mobile user m. We determine the set Q of quadrants to be considered based on the steady motion density function $p(\theta)$ of the mobile user m (recall Figure 10(a)). $|Q|$



(a) Corner Points (b) Revising (c) Revised Corner Points (d) Dominant Points (e) Patch and Trim (f) Final AFR

Fig. 13. Computing a Motion-aware AFR

is at most 2. If θ is 100° and we use $p(\theta)$ with $z = 8$, then the range of moving direction will be between 78.5° and 122.5 °. The range of direction overlaps with Quadrant I and II. Therefore we consider two quadrants out of four. If the range of moving direction overlaps with only one quadrant (e.g., when θ is 30°), then only one quadrant is relevant and selected in this step.

Step 2: Find Candidate Corner Points. Let A be the empty region in which m resides. In each selected quadrant, we first use the the function REGIONSEARCH to find all neighboring empty regions of A, which are included in this quadrant. Then we examine each of neighboring empty regions, and add its four corner points into the set of candidate corner points. This set of candidate corner points will be used to determine the component rectangle of the extended AFR in the selected quadrant. In our running example, corner points in the quadrants I and II are represented by hollow circles as shown in Figure 13(a).

Step 3: Find candidate component rectangle by revising corner points. For each selected quadrant, we examine all candidate corner points and revise those that may not form a component rectangle of the extended AFR. A component rectangle of the extended AFR is the empty rectangle region that has o, the current position of m, as one of the corner points. For example, we examine the four corner points of B in the left top quadrant as shown in Figure 13(a). Consider a component rectangle consists of p_1 and o. This component rectangle overlaps with the spatial alarm S_1 as shown in Figure 13(b). Therefore the corner point p_1 needs to be moved to q_1 so that it avoids to overlap with S_1. Similary, a corner point p_4 should be revised to move to q_4 which has the same x-value as S_1 's right border. The new component rectangle formed with o and q_2 as two diagonal corner points will not overlap with any spatial alarms. The same process runs iteratively until every candidate corner point is examined and revised. The revised corner points are shown by dashed arrows in Figure 13(c).

Step 4: Find Dominating Points. Let Quadrant I denote the upper right quadrant, Quadrant II denote the upper left quadrant, Quadrant III denotes the bottom left quadrant, and Quadrant IV denotes the bottom right quadrant. In each quadrant we change the meaning of the dominant points for $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$ as follows:

(a) Quadrant I: p_1 dominates p_2 if $x_1 \geq x_2$ and $y_1 \geq y_2$
(b) Quadrant II: p_1 dominates p_2 if $x_1 \leq x_2$ and $y_1 \geq y_2$
(c) Quadrant III: p_1 dominates p_2 if $x_1 \leq x_2$ and $y_1 \leq y_2$
(d) Quadrant IV: p_1 dominates p_2 if $x_1 \geq x_2$ and $y_1 \leq y_2$.

Based on the definition above, the set of dominating points are represented as solid black dots in Figure 13(d), namely d_{21}, which disregards two hollow points with the same x values, d_{22}, which disregards three hollow points, d_{11}, which disregards one hollow point with the same x-value, d_{12}, which disregards two hollow points, one with the same x-value and the other with the same y-value; and d_{13}.

All the hollow dots are dominated by the black dot corner points. An important property of a dominating point is that the size of the component rectangle defined by the current location o of m and a given black dot corner point is maximized, compared to the component rectangle defined by the current location o of m and a hollow corner point dominated by the given black dot. For example, the component rectangle with a dominant black dot is larger than the component rectangle with a hollow corner point.

Step 5: Patch and Trim Component Rectangles. The set of dominating points form corners of component rectangles in each quadrant. The final AFR is composed by patching one component rectangle from each quadrant and trimming the patched rectangle so that the distance from the current location to the border of resulting AFR rectangle is maximized while m heading towards the θ direction. In Figure 13(e), there are five component rectangles, each corresponds to one of the five dominating points marked in black dot. In quadrant II, we select the component rectangle with o and d_{21} instead of the one with o and d_{22} because the former provides the longest distance to the border. Similarly, in quadrant I we choose the component rectangle with o and d_{11}, because the distance to the border remains the longest. If we choose d_{12}, then the final AFR is wider but shorter and the distance to the border is shorter.

The time complexity for computing motion-aware AFR is $O(n^2)$, given that Step 1 finishes in $O(1)$, step 2 in $O(n \log n)$, step 3 in $O(n^2)$, step 4 in $O(n \log n)$ by divide and conquer, and step 5 takes $O(n)$.

5.3. Distributed Mondrian Index

Given n mobile users subscribing to public and private spatial alarms, there are three alternative ways of creating and maintaining Mondrian indexes. First, we can create and maintain a single Mondrian tree for all mobile users and all their spatial alarms. We call it the *centralized* approach. Alternatively, we can create n Mondrian tree indexes, each is devoted for one mobile subscriber, which indexes all alarms subscribed by this subscriber, including public, private, and shared alarms. We call it the *distributed* approach. The third alternative is to create one Mondrian index for all public alarms, and n Mondrian tree indexes for all private and shared alarms, each is dedicated to one of the n mobile subscribers. We call it the *Hybrid* approach.

Given the specific characteristics of spatial alarms, the approaches of creating and maintaining individual Mondrian trees, one per client, can significantly minimize the overhead of searching for relevant alarms and AFRs of a given mobile user.

For example, if Alice installed 10 spatial alarms and Bob installed 30 alarms, then we create a single Mondrian tree with 40 alarms under the centralized approach and two Mondrian trees, one for Alice with 10 alarms and the other for Bob with 30 alarms, under the distributed approach. Then, the average size of AFRs in the centralized approach will be much smaller than that of AFRs in the distributed approach because the centralized approach inserts more alarms in a Mondrian tree. Furthermore, the less nodes we have in a Mondrian tree, the faster it takes to find a leaf node because the search conducted by Alice will not be affected by the spatial alarms installed by another user Bob. In addition, three alternative system architectures can be used to support spatial alarm processing: server-centric, client-centric, and distributed client-server. Considering that the client-centric architecture is only applicable for processing private spatial alarms [Murugappan and Liu 2008], we below focus on server-centric architecture and distributed client-server architecture.

In the server-centric architecture, spatial alarms will be installed, subscribed and processed at the server and mobile clients do not contribute directly to the spatial alarm processing tasks. Mobile clients only receive alarm notification when entering their alarm target regions.

In the distributed client-server architecture, the server creates and maintains one Mondrian index per mobile subscriber. Insertion of new public or shared alarms will trigger the server to broadcast the newly installed alarms to those mobile clients whose subscriptions match with this new alarms, so that each mobile client can insert the newly added alarm into its local Mondrian tree. Insertion of a private alarm only involves the insertion of this alarm to the local Mondrian tree of its owner. A spatial alarm is removed from the system (client and server) upon reaching its expiration time. Alarm expiration is checked at each alarm evaluation. Spatial alarm processing is accomplished by the server and the mobile clients collectively. Several strategies can be used for partitioning of spatial alarm processing tasks into server side and client side processing. We below describe three possible strategies for implementing the Mondrian tree approach under the distributed client-server architecture.

The first strategy will have the sever perform the following four tasks: (1) construct and maintain the Mondrian tree for each mobile object m; (2) search the Mondrian tree index to find AFR_m for m; and (3) compute AFP_m for m; and (4) send AFP_m to m. In this scenario, the client application checks if AFP_m expires. If so, it sends a new AFP_m request message to the server.

The second strategy will have the server perform only the first two tasks and a modified version of the 4th task. Concretely, the server sends the current AFR_m to m instead of the AFP_m. Now the client computes AFP_m locally using AFR_m (task 3). The client only reports to the server when it moves outside of AFR_m or an alarm monitoring region.

The third strategy will have each client build a Mondrian tree. Each client lookups the index locally, finds its current AFR, and computes AFP accordingly. A client only reports to the server if it arrives at an alarm monitoring region. All public, private and shared alarms are installed at the server and distributed to clients by the server.

The choice of which strategy to use depends primarily on the capacity of mobile clients. Some clients have limited capability in terms of battery power, computing and storage capacity, whereas others are equipped with computing and memory capability equivalent to a laptop computer such that it can store the Mondrian tree that indexes all of its subscribed alarms locally. In this situation, a mobile subscriber who is capable of storing its own Mondrian tree locally can also perform the Mondrian tree lookup and AFP computation locally. Thus this strategy significantly reduces the amount of client to server communication cost.

6. EXPERIMENTS AND EVALUATION

In this section we report our experimental evaluation of the performance and effectiveness of the Mondrian tree approach to spatial alarm processing. Our experiments are conducted with two objectives. First, we want to compare Mondrian index with a set of popular spatial indexing techniques, such as R-tree, k-d tree, and Quadtree in terms of multiple parameters, including total time, index lookup time, memory size, AFR computation time, average size of AFR, and average client-to-server communication messages. Our experimental results show that Mondrian tree approach significantly outperforms R-tree, k-d tree, and Quadtree for spatial alarm processing, due to faster AFR computation, longer AFP, and indexing of both alarms and AFRs with Mondrian trees. Second, we conduct experiments to compare different design choices for processing spatial alarms using Mondrian tree indexes, including (i) comparing the basis AFP (alarm free period) and the steady motion based AFP with respect to periodic evaluation; (ii) comparing the basic Mondrian tree with Mondrian-PAT and Mondrian-MPAT in terms of total time, AFR computation, size of AFR, the number of AFR crossing checks required, and average AFP interval length; (iii) comparing centralized, distributed, and hybrid approach to creating and maintaining Mondrian

indexes in terms of AFR computation and average size of AFR , and (iv) comparing basic Mondrian index and Mondrian* index in terms of memory, average depth of the trees, average number of nodes per tree, and search time.

6.1. Experiment Setup

We extend GTMobiSim mobility simulator [Pesti et al. 2009] and generate a set of traces of moving vehicles on a real world road network. The road network data are obtained from the National Mapping Division of the U.S. Geological Survey[USG] in the form of Spatial Data Transfer Format[SDT]. Vehicles are distributed randomly on the road segments according to traffic densities determined from the traffic volume data [Gedik and Liu 2005]. These vehicles move on the roads of metro Atlanta. The number of spatial alarms and the number of vehicles vary from 1,000 to 10,000. The default number of vehicles is 5,000 and the default number of alarms is 5,000.

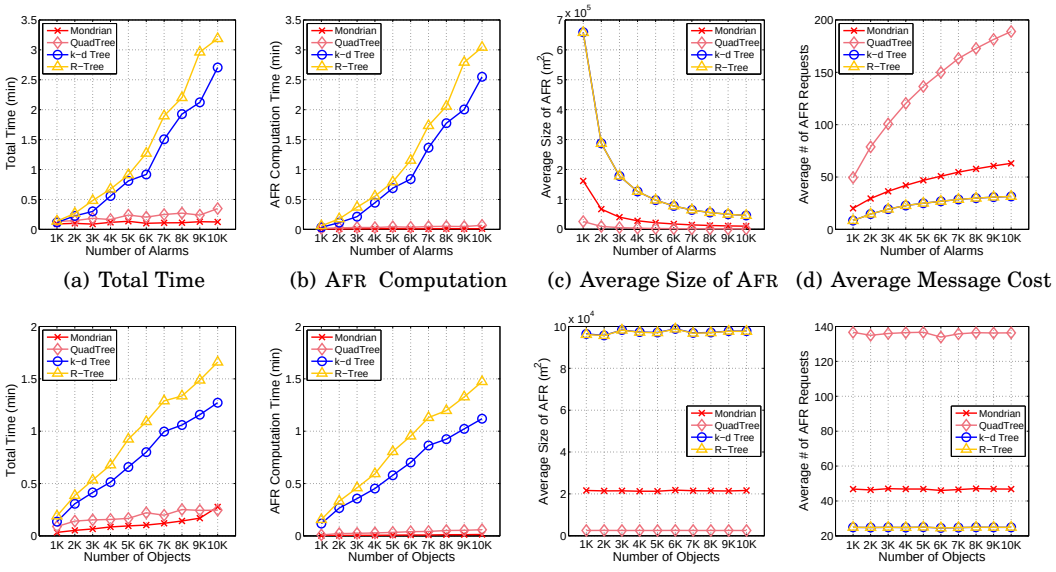
Given that Mondrian tree is a memory based space partitioning index structure, we modified other data structures so that they stored data in the main memory.

For R-tree, Quadtree, and k-d tree, the index structure does not provide a good quality of AFR. In this experiments we compute AFRs by the safe region algorithm in [Bamba et al. 2008] for distributed processing of spatial alarms. It consists of two steps. First, it finds n nearest alarms and computes the AFR using a greedy approach in $O(n^2)$. The parameter n is a system supplied parameter. When n is too small, AFR is too small. On the other hand, if n is too big, the AFR computation is too expensive.

All the Mondrian trees used in this experimental evaluation are centralized Mondrian tree except in the experiment for performance of multiple Mondrian trees.

All experiments run on a Linux machine with Intel Core 2 Duo CPU 2.8GHz and 4GB RAM.

6.2. Effectiveness of Mondrian Tree Index



(a) Total Time (b) AFR Computation (c) Average Size of AFR (d) Average Message Cost

(e) Total Time (f) AFR Computation (g) Average Size of AFR (h) Average Message Cost

Fig. 14. Performance Results in Server-side

The first set of experiments is designed to compare the Mondrian tree against other data structures in spatial alarm processing by measuring total time, index lookup time, memory size, AFR computation time, average size of AFR, and average client-to-server communication messages. In order to understand how the number of alarms and the number of users impact the alarm process performance, we design two groups of experiments, one with varying number of users from 1,000 to 10,000 and 5,000 alarms (Figures 14(a) - (d)) and the other with varying the number of alarms from 1,000 to 10,000 and 5,000 vehicles (Figures 14 (e) (h)). Note that to be fair in the comparison of centralized Mondrian tree with other index structures, all experiments reported in this first set is using the basic Mondrian tree without AFR or AFP optimizations.

Figures 14(a) and 14(e) shows total time spent for alarm processing for varying number of alarms and varying number of users respectively. In both figures, although the total time increases for all approaches as the number of alarms or users increases, the rate of increase for Mondrian tree is significantly smaller comparing with R-tree and k-d tree. The reason Quadtree has slightly slower than Mondrian tree is that compared to other two data structures it only concerns a small leaf node and a set of spatial alarms in it, which reduces the number of alarms to be considered for computing AFRs.

Figures 14(b) and 14(f) shows total AFR computation time. By comparing with Figures 14(a) and 14(e), we note that AFR computation takes up to 90% of total alarm processing time. On each AFR request by mobile user m, the server looks up spatial alarms with regard to m's current location. Given a location, Mondrian tree approach does not compute AFRs. It just looks up the relevant leaf node. However, R-tree and k-d tree need to dynamically compute AFRs upon each user request in $O(n^2)$, so their AFR computation costs are quadratic in comparison with Mondrian approach.

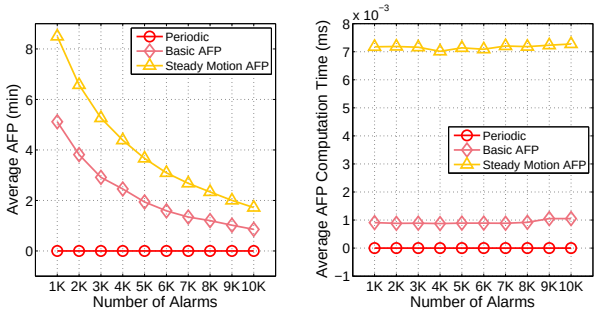
Figures 14(c) and 14(g) compares the average size of AFRs. The Mondrian tree partitions the universe of discourse into relatively smaller size of rectangles on average and thus smaller AFRs. Therefore the average size of AFRs in the Mondrian tree is smaller than one in R-tree or k-d tree as shown in 14(c). Like Mondrian tree, Quadtree is also a region partition tree. Therefore the average size of AFR is also small. As we increase the number of alarms, the average size of AFRs decreases because more empty regions are occupied by alarms. However, it is obvious that the average size of AFRs does not change much while fixing the number of alarms and varying the number of users as shown in Figure 14(g).

The message cost from client to server is shown in Figures 14(d) and 14(h). Due to the smaller size of AFR in Mondrian tree, the average number of client-to-server message for requesting AFRs is slightly larger than other data structures. Quadtree also indexes smaller empty regions, but once it enters a region that has a spatial alarm, then it needs to compute AFR. This makes users using Quadtree frequently cross the border and compute AFRs. Therefore it has the biggest number of message cost as shown in Figure 14(d). When we vary the number of mobile objects but fix the number of alarms in Figure 14(h), the number of messages requesting AFRs barely changes because average size of AFRs are similar for all mobile objects, and thus each has similar probability of crossing AFRs.

6.3. Effectivenss of AFP

The second set of experiments compared the two AFP approaches against periodic alarm evaluation approach. Figure 15(a) shows that the basic AFP and the steady-motion AFP have much longer AFP interval compared to periodic approach and thus reducing the amount of unnecessary alarm evaluations. Also steady-motion AFP outperforms the basic AFP , though the difference is decreasing as the number of alarms increases, because more alarms implies smaller AFRs. Figure 15(b) shows that the average amount of time for computing AFP is less than $1ms$, compared to the average

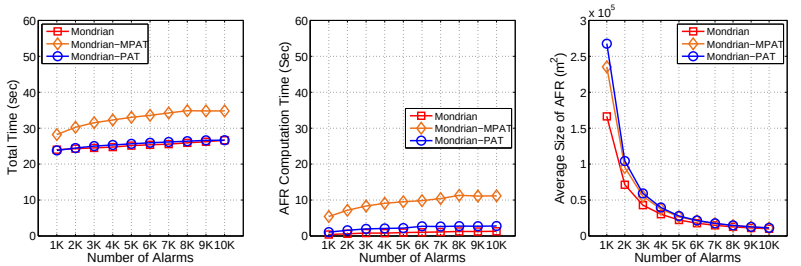
AFP interval of 8 minutes and 30 seconds for 10,000 alarms. The experiment is the simulation of 30-minute driving. If a user actually moves for 30 minutes as the simulation, the spatial alarm processing can enter hibernation for about 8 minutes and 30 seconds, 28% of 30 minutes of driving. Also Figure 15(b) shows that although steady motion AFP has the longest AFP interval, it takes more time to compute AFP because it needs to compute the average of AFPs for more than one direction.



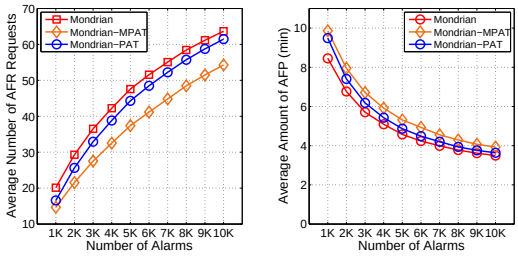
(a) Average AFP (b) AFP Computation Time

Fig. 15. AFPs vs. Periodic Evaluation

6.4. Effectiveness of AFR Optimizations



(a) Total Time (b) AFR Computation (c) Average Size of AFR



(d) Average Message Cost (e) Average Length of AFP

Fig. 16. Performance of AFR Optimizations

Figure 16 shows the comparison of the basic AFR with the two AFR optimizations. In this set of experiments, **Mondrian** refers to the use of empty region from the basic Mondrian tree construction as AFRs (no optimization). **Mondrian-MPAT** refers to Mondrian approach powered with MPAT, and **Mondrian-PAT** is the Mondrian approach powered by PAT. In all cases, we compute AFP based on AFRs for processing spatial alarms.

Figures 16(a) and 16(b) measure the total time and AFR computation time with varying number of alarms from 1,000 to 10,000. We can see that up to 28% of time is used for AFR computation. Comparing to **Mondrian**, **Mondrian-PAT** takes slightly more time to patch and trim than **Mondrian**. **Mondrian-MPAT** takes the longest time because it performs series of operation to compute an extended motion-aware AFR.

Figure 16(c) compares the basic Mondrian with optimized Mondrain in terms of the size of AFR. We see that the size of AFR in **Mondrian-PAT** is on average larger than in the size of AFR in **Mondrian-MPAT**. This is because motion-aware PAT enables us to disregard about a half of empty regions that are irrelevant for a given mobile user. However, as shown in Figure 16(d), the message cost of **Mondrian-MPAT** is smaller than **Mondrian-PAT** because **Mondrian-MPAT** considers moving direction and extends an AFR along the moving direction. Furthermore, extended AFRs based on **Mondrian-MPAT** also provide longer AFP as shown in Figure 16(e).

6.5. Performance of Distributed Mondrian Trees

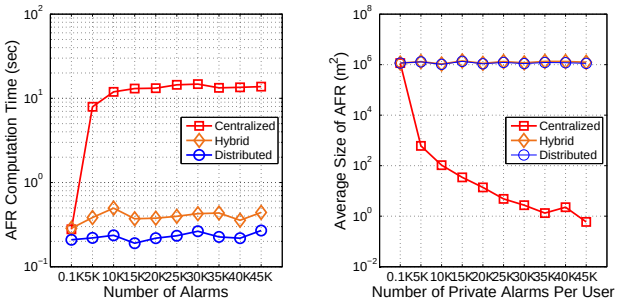
In this set of experiments, we compare the performance of distributed Mondrian trees against the performance of centralized Mondrian index and hybrid Mondrian index. We set the number of users to be 5,000, the number of public alarms to be 100, and increase the number of private alarms per user from 0 to 10. Therefore the total number of alarms vary from $100, (5,000 + 100), (10,000 + 100), \cdots, (50,000 + 100)$.

Figure 17(a) shows that **Centralized Mondrian tree** takes the longest time for computing AFR. If we add one additional private alarm to for all users, then **Centralized** Mondrian index needs to add 5,000 (1 alarm $\times$ 5,000 users) more alarms in the tree. On the other hand, the **Hybrid** and **Distributed** Mondrian indexing increase only 1 alarm. Similarly, when we add 10 additional private alarms per user, then **the Centralized Mondrian** will add 50,000 (10 alarms $\times$ 5,000 users) more alarms. Therefore **Centralized Mondrian** has the worst performance. When there are 100 percent of public alarms, then there is no difference in each approach. **Hybrid** needs to check two trees: one for public alarms and the other for private alarms. Hence it takes more time than **Distributed**.

Figure 17(b) shows that **Centralized Mondrian approach** has the smallest average AFR size. On 100 percent public alarms, the size of AFR is the same because each approach has the same alarms. When we assign one additional private alarms to each user, then **Centralized Mondrian** will have 5,000 more alarms while distributed or Hybrid Mondrian approaches will an increase by only 1 alarm. Therefore as we increase the number of private alarms per user, the average size of AFR in the **centalized** Mondrian tree decreases dramatically.

6.6. Effectiveness of Mondrian*

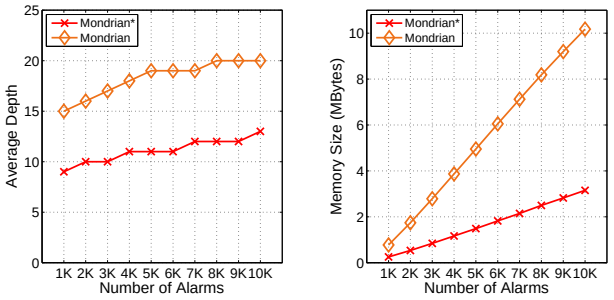
We presented Mondrian*, an optimized Mondrian tree, in Section 4. We performed a set of experiments to measure the performance of Mondrian* against Mondrian basic in terms of memory size and average depth of the tree. Figure 18(a) shows the average depth of Mondrian and Mondrain* by varying the number of spatial alarms from 1K to 10K. Clearly, Mondrian* has the shorter depth on average for a couple of reasons. First, the number of children nodes that a non-leaf node may have in Mondrian* tree is four,



(a) AFR Computation (b) Average Size of AFR

Fig. 17. Server-centric vs Distributed Approach

whereas it is two in the Mondrian tree. Second, in batch construction of Mondrian* tree, it chooses an alarm that is approximately located in the center of all alarms as the next alarm to be inserted so that we can distribute remaining alarms into the four children evenly. Figure 18(b) shows the memory size. Recall Figure 5(d), Mondrian tree needs 9 nodes for one spatial alarm as shown in Figure 5(d) while Mondrian* tree only needs five as shown in Figure 7(b). Hence, Mondrian* tree takes shorter time to lookup and needs less memory.



(a) Average Depth (b) Memory Size

Fig. 18. Mondrian* vs. Mondrian

7. CONCLUSION

In this paper, we have presented the design and implementation of the Mondrian tree index, a fast index structure for scalable processing of spatial alarms. Compared with conventional spatial indexes, such as R-tree, Grid, and k-d tree, the main distinguishing feature of Mondrian tree, is that the Mondrian tree approach indexes not only spatial alarms but also empty regions, which enables us to look up AFRs fast compared to other data structures. Another novelty of the Mondrian tree index is its ability to utilize the characteristics of spatial alarms to create and maintain one Mondrian tree for each mobile subscriber, which is particularly effective when there is relatively small number of public alarms compared to the total number of private alarms in the system. We also provide a set of optimization techniques for scaling spatial alarm processing based on Mondrian index, such as motion-aware AFP extended AFRs using

PAT and MPAT. Our experiments show that Mondrian tree can dramatically minimize the amount of unnecessary AFR and scale the spatial alarm processing, compared to R-tree, Grid, and k-d tree.

REFERENCES

Spatial data transfer format. http://www.mcmcweb.er.usgs.gov/sdts/.

Us geological survey. http://www.usgs.gov/.

BAMBA, B., LIU, L., YU, P. S., ZHANG, G., AND DOO, M. 2008. Scalable processing of spatial alarms. In *Proc. HiPC*.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. In *SIGMOD*.

CHAZELLE, B., DRYSDALE, R., AND LEE, D. 1984. Computing the largest empty rectangle. In *STACS 84*.

FINKEL, R. A. AND BENTLEY, J. L. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*.

GEDIK, B. AND LIU, L. 2004. Mobieyes: Distributed processing of continuous moving queries on moving objects in a mobile system. In *LNCS*.

GEDIK, B. AND LIU, L. 2005. Location privacy in mobile systems: A personalized anonymization model. In *Proc. IEEE ICDCS*.

GROVE, J. V. 2010. Morbile marketer. http://mashable.com/2010/08/18/shopalerts.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*.

HASAN, M., CHEEMA, M., LIN, X., AND ZHANG, Y. 2009. Efficient construction of safe regions for moving $k$nn queries over dynamic datasets. In *Advances in Spatial and Temporal Databases*. Springer.

HENRICH, A., SIX, H.-W., AND WIDMAYER, P. 1989. The lsd tree: spatial access to multidimensional point and non-point objects. In *Proc. VLDB*.

HU, H., XU, J., AND LEE, D. L. 2005. A generic framework for monitoring continuous spatial queries over moving objects. SIGMOD.

MURUGAPPAN, A. AND LIU, L. 2008. Energy-efficient processing of spatial alarms on mobile clients. In *Proc. ICSDE*.

NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. 1984. The grid file: An adaptable, symmetric multikey file structure. In *TODS*.

PESTI, P., BAMBA, B., DOO, M., LIU, L., PALANISAMY, B., AND WEBER, M. 2009. Gtmobisim: A mobile trace generator for road networks. Tech. rep.

PESTI, P., LIU, L., BAMBA, B., IYENGAR, A., AND WEBER, M. 2010. Roadtrack: Scaling location updates for mobiles on road networks with query awareness. In *VLDB*.

PRABHAKAR, S., XIA, Y., KALASHNIKOV, D. V., AREF, W. G., AND HAMBRUSCH, S. E. 2002. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. In *IEEE Transactions on Computers*.

SEEGER, B. AND KRIEGEL, H. P. 1988. Techniques for design and implementation of efficient spatial access methods. VLDB.

SOHN, T., LI, K. A., LEE, G., SMITH, I., SCOTT, J., AND GRISWOLD, W. G. 2005. Place-its: A study of location-based reminders on mobile phones. UBICOMP.

Received February 2007; revised March 2009; accepted June 2009