Efficient Constraint Processing for Location-aware Computing

Zhengdao Xu and Hans-Arno Jacobsen
Department of Computer Science and
Department of Electrical and Computer Engineering,
University of Toronto
10 King's College Road, Toronto, Ontario, Canada, M5S 3G4
zhengdao@cs.toronto.edu, jacobsen@eecg.toronto.edu

ABSTRACT

For many applications, such as friend finder, buddy tracking, and location mapping in mobile wireless networks or information sharing and cooperative caching in mobile ad hoc networks, it is often important to be able to identify whether a given set of moving objects is close to each other or close to a given point of demarcation. To achieve this, continuously available location position information of thousands of mobile objects must be correlated against each other to identify whether a fixed set of objects is in a certain proximity relation, which, if satisfied, would be signaled to the objects or any interested party. In this paper, we state this problem, referring to it as the *location constraint matching problem* and present and evaluate solutions for solving it. We introduce two types of location constraints to model the proximity relations and experimentally validate that our solution scales to the processing of hundreds of thousands of constraints and moving objects.

Categories and Subject Descriptors

H. [**Information Systems**]; H.4 [**Information Systems Applications**]: Location-based services; H.3.3 [**Information Search and Retrieval**]: Information filtering

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Location-based Service, Location Constraint, Constraint Processing, Space-partitioning k-d-tree, Cost Model

1. INTRODUCTION

With the advances in wireless communication and location positioning technology [36], the potential for tracking, correlating, and filtering information about moving objects has greatly increased. For example, it has become possible to track the location of mobile users in a wireless network or in a building [36, 31], the discrete location of vehicles or packages in delivery [1] and even the movement of livestock or fish for environmental purposes [2].

An important problem in the context of applications that leverage this tracking potential is how to efficiently determine whether for any given number of sets of moving objects, the objects per set are *close to* one another or *close to* a given point of demarcation. We refer to this problem as the *location constraint matching problem*. In a wireless network, for example, an operator may want to offer alerting services that notify members of a group (e.g., a group of friends or family), if they are *close to* each other or *close to* a designated point in the environment (e.g., the CN Tower in Toronto). In a goods and packages delivery chain, the operator may want to know if a set of packages routed to the same or similar destinations are *close to* one another and may benefit from common delivery to amortize delivery cost. In ad hoc networking, given a set of mobile devices, the devices may want to know if they are *close to* each other to benefit from this proximity for resource sharing (e.g., content, communication, caching, and computation.)

To model this *close-to-relation* among a set of n moving objects and a set of moving objects and a point of demarcation, we introduce two types of constraints, the *n-body constraint* and the *n-body static constraint*. We refer to these constraints as *location constraints*, as they are defined over the location position of the objects. The location constraints are formally defined as follows:

1. The *n-body constraint* is of the form $|{p_1}^t, {p_2}^t, ..., {p_n}^t| \leq d$. It is satisfied if the n moving objects, identified by, $p_1, p_2, ..., p_n$, can be enclosed by a sphere with diameter d at some time, t. p_i $(1 \leq i \leq n)$ is the identifier of object i. In our notation ${p_i}^t$ is interpreted as the coordinate of object i at time t. d is referred to as the *alerting distance*.

2. The *n-body static constraint* is of the form $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t| \leq d_A$, where A is the coordinate of some static point. It is satisfied if the n moving objects, identified by, $p_1, p_2, ..., p_n$, are within the given range, d_A, of the static point A at some time, t. Here d_A is referred to as the alerting distance.

The location constraint matching problem can then be stated as follows: Given a set of location constraints $C = \{c_1, c_2, ..., c_k\}$, which designate the desired location relationship among a set of m, possibly, moving objects $P = \{p_1, p_2, ..., p_m\}$, *continuously* determine all constraints c_i in C that are satisfied. The location

MDM 2005 05 Ayia Napa Cyprus

constraints are continuous queries that once submitted to the system remain active until explicitly revoked.

Existing data management and indexing techniques for moving objects [22, 16, 27, 28, 29] are well suited to support range query and indexing of large data sets. However, these techniques are not suited to solve the location constraint matching problem, because they do not address the efficient evaluation of constraint-based correlations among different sets of objects. In a wireless network, often thousands of moving objects could be involved in location constraint matching. If these objects continuously move and frequently change location vectors, the underlying database would be busy updating the objects' location without being able to evaluate the location constraints defined among the objects of different sets of objects.

The location constraint matching problem is also different from the nearest neighbor problem [35, 15]. The nearest neighbor problem determines the nearest object(s) to a given point among all the objects in the space. However, the location constraint matching problem determines whether a specific set of objects are in a given spatial constellation to each other at one point in time. Our work mainly focuses on how to evaluate a large number of constraints in different proximity relations efficiently; not on how to solve a single location constraint.

Range query techniques may be able to evaluate a single static constraint over available location data, however, the stated problem involves many different constraints that have to be re-evaluated as the location of the involved objects changes, which is the rule, not the exception in our scenarios. A range query-based approach to static location constraint matching would require the evaluation of one range query per registered static constraint as new location data becomes available. While this is possible, we intend to prune the number of location constraints that need to be evaluated by ruling out constraints that cannot be satisfied and thus increase evaluation performance, as compared to an approach that individually evaluates each constraint. Range queries have mostly been designed to evaluate one query over large amounts of data. The location constraint matching problem is characterized by a need to evaluate a large number of queries (i.e., constraints) based on the changing location information of one object involved. Furthermore, for the case of non-static n-body constraints, range queries do not apply.

In our design, we assume that the server running the location constraint matcher periodically receives the objects' location updates. These updates trigger the constraint evaluations and yield, as result, the matching constraints. In practice location position information could be either obtained through GPS, network-enhanced GPS, ground-based sensors (e.g., networks of RFID readers or even surveillance cameras) or other location positioning technology [36]. A detailed description of a fully functional software demonstration of our constraint processing system is described in Xu and Jacobsen [33]. Further details on the use of our algorithms for location position correlation in a real cellular network are described in a companion paper in Xu and Jacobsen [34].

The contributions of our work are as follows:

1. We state the location constraint matching problem, both for n-body location constraints and n-body static location constraints. These constraints model a *close-to-relation* among a set of n moving objects as well as a set of n moving objects and a static point, respectively.
2. We develop efficient algorithms based on space partitioning to solve the location constraint matching problem for both types of constraints.
3. We develop an analytical model to quantify the cost of the evaluation for both main memory-based and I/O-incurring environments.
4. We experimentally validate the efficiency of our algorithms with various partition schemes and movement patterns and demonstrate the algorithms' scalability.

In the next section, we describe the location constraint matching algorithms. In Section 3, we develop the analytical model to asses the constraint evaluation cost, determine system parameters, and estimate secondary storage access cost. Section 4 presents the experimental evaluation of the algorithms. Section 5 briefly describes the overall system architecture of our location constraint matching engine. In Section 6 we put our work in perspective to related approaches.

2. MATCHING ALGORITHM

Our solution is based on point index schemes, such as the k-d-tree [5, 6] and the grid index [21]. Traditional k-d-tree indexes and grid indexes have been used to support orthogonal range search in databases. We extend these indexing schemes to index the continuously changing location of moving objects, and base an efficient solution to the location constraint matching problem on them. The intuition behind our algorithms is that, a space partitioning that approximates the location of the moving objects, rather than the exact position of each object, is sufficient for most constraint evaluations. The details will be illustrated in the following subsections. Below we describe our space partition schemes and show how they benefit our approach for efficient location constraint evaluation.

2.1 Space Partitioning and Partition Update

Different from the traditional k-d-tree, the k-d-tree variant we use partitions the space rather than the objects in the space; we call this variant the space-partitioning k-d-tree (SPKDT). Fig. 1 shows how a 2-d square space is partitioned and shows its corresponding k-d-tree. The root of the tree is used to represent the whole query space. It is extended by splitting the whole query space into two sub parts and the description of the whole space and the splitting line L_1 is stored in the root node of the tree. Then at the two children of the root, which represent one sub space each, the partition process continues recursively using the horizontal or vertical splitting lines, until a sufficient granularity is reached. In SPKDT, the partitions created do not have to be equally sized; the horizontal and vertical splitting lines do not have to alternate strictly as the tree extends. From the tree construction, each leaf node represents one partition of the whole space and every internal node stores the information of the splitting line (or hyper plane for higher dimensions).

We call it a *partition update* when an object moves across the boundary of the partition it was last located in. Even though the movement of objects is (mostly) continuous in reality, the location position information for each object only captures the discrete position of objects periodically. In order to track the mobile objects during the partition update and to make our algorithm function properly, each object has to be associated with the correct leaf node as it changes its position. To make this efficient, the algorithm first

checks if the moving object is still in the partition where it was last located. If not, the algorithm backtracks level-by-level up the tree until it finds the splitting line the object has not yet crossed. Then the object is inserted from that node down to its current partition (a new leaf node). The backtracking partition update closely simulates the near-to-far search. As long as objects show up in the same or nearby partitions with high probability in the next location update, this backtracking update greatly reduces the update cost as compared with a naïve approach, which inserts the objects from the root down to the leaf node. This effect becomes more severe when the tree is very high. The assumption that the objects show up in the same or nearby partitions with high probability in the next location update step is reasonable since the movement of the objects is continuous with some reasonable speed and the size of the partitions can be customized. The partition update can be triggered either on the server side or on the object side. Triggering the update on the object's side assumes that the moving object knows the partition boundaries and can correlate its position information with these boundaries.

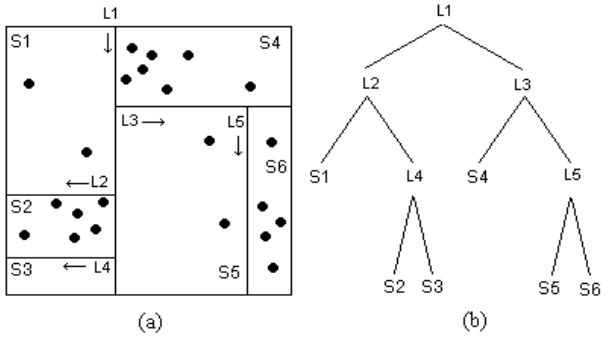

Figure 1: Space Partitioning: (a) 2-d Partition, (b) k-d-tree of the Partition

An alternative method as point index is the grid index [21], in which the whole space is partitioned into a grid of equal-sized cells and the new grid that accommodates the point at position $[x_1, x_2, ..., x_n]$ is computed as $Grid[\lfloor x_1/w_1 \rfloor, \lfloor x_2/w_2 \rfloor, ..., \lfloor x_n/w_n \rfloor]$, where the grid is of size $w_1 \times w_2 \times ... \times w_n$. One nice property of the grid index is that the mobile objects can be associated with the grid in constant time, However, since the size of the partitions has to be the same, the flexibility of the space partitioning for solving our problem is compromised.

2.2 Evaluation Algorithms

With the above pre-processing (i.e., the construction of the SPKDT or the grid and the partition update for moving objects), the evaluation of location constraints can be performed very efficiently. We refer to the algorithm evaluating n-body constraints as the n-body matching algorithm (NB) and the algorithm for evaluating the n-body static constraints as the n-body static matching algorithm (NBS).

Our constraint evaluation algorithms take advantage of the space partitioning and use the partition information as a rough estimate for the position of the moving objects. For certain constraint evaluations, this approximation information is enough to determine a result. In the following, we present our evaluation algorithms in detail.

n-body Constraint Evaluation (NB): To evaluate n-body location constraints, $|{p_1}^t, {p_2}^t, ..., {p_n}^t| \leq d$, our algorithm first classifies the constraints into three categories: those *unsatisfied* (Class A), those *satisfied* (Class B) and those still *uncertain* (Class C). The classification is based solely on the knowledge of the partitions that accommodate the moving objects, $p_i, (i = 1, ..., n)$; so the classification does not change as long as the objects remain in their partitions (even if the objects update their location within their partitions.)

The classification is invoked every time some object changes its partition (not for every location update). When some object, p_i, moves into a new partition S_i, all the constraints it is associated with need to be re-classified. Suppose that c_j is the constraint p_i is associated with and $P = \{p_1, p_2, ..., p_n\}$ is the set of bodies involved in c_j ($p_i \in P$). Then, if there exists some p_k ($p_k \in P$) in the partition S_k ($k \neq i$) and the minimum distance between S_k and S_i is larger than d ($alerting\ distance$), c_j cannot be satisfied (Class A). On the other hand, if all bodies in P are in the partitions whose union can be covered by a circle with d as the diameter, c_j must be satisfied (Class B). If c_j does not fall into the above two categories, the result of the constraint is uncertain (Class C) and in order to get the diameter of the smallest enclosing disk the explicit computation is needed. Fig. 2 provides examples illustrating this classification algorithm. The function `Classify`(c, $type$) classifies the constraint c to the class specified by $type$. The function `DiskSize`($region$) computes the diameter of the smallest circle enclosing the space specified by $region$. In the `DiskSize` function, the computation of the smallest enclosing disk for the union of partitions is reduced to the computation of the smallest enclosing disk for the vertices of those partitions.

procedure *Classify_Constraints_NB*(MobileObject p)
/* This function is called when some object p updates its */
/* partition. It classifies the constraints p is associated with */
begin
  for each Constraint c that p is associated with
  {
    let $P = \{p_1, p_2, ..., p_n\}$ be the set of bodies in c;
    if ($\exists\ p_i \in P \wedge c.d < Distance(p.parititon, p_i.partition)$)
      Classify(c, Class_A);
    else if (DiskSize($\bigcup_{i=1}^{n} p_i.partition$)$< c.d$)
      Classify(c, Class_B);
    else
      Classify(c, Class_C);
  }
end

Figure 2: Classification Algorithm (NB)

Fig. 3 illustrates how the classifications are managed. For example, constraint $|a, b, c, e| < 3$ is classified as Class A, unsatisfied, because the minimum distance between the partition S_{11} (where a is located) and partition S_{55} (where e is located) is $3\sqrt{2}$, which is larger than the alerting distance 3. This constraint cannot be satisfied unless at least one body involved changes its partition, and therefore invokes another classification.

After the classification, the evaluation algorithm only considers constraints of Class C. For computing the smallest enclosing disk of the n objects (or partitions), the algorithm in [32] is used; it computes the smallest circle that encloses n points in $O(n)$ time.

n-body Static Constraint Evaluation (NBS): To evaluate n-body

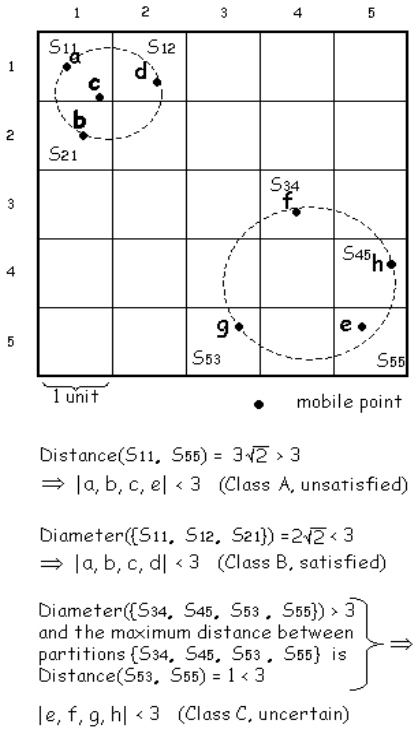

Figure 3: Classification of n-body Constraint

static constraints $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t| \leq d_A$, we observe that the constraint is satisfied if and only if all the p_i ($i \in n$) are in the circle with static point A as the center and d_A as the radius. Depending on whether the partition is inside, intersecting, or outside the boundary of that circle, we can determine the internal, bounding, and external partitions to the region with the static point A as the center and d_A as the radius. The distance between the moving object and static point A can be tracked according to the type of the partition the object is in. The distance between A and the objects inside the internal partition is smaller than d_A and the distance between A and objects inside the external partition is greater than d_A. The explicit distance computation is only needed for the moving object inside the bounding partition.

The partition update and constraint classification is similar to the NB algorithm. If some object is in the external partition, the constraint cannot be satisfied (Class A); if all objects are in the internal partitions, the constraint must be satisfied (Class B); if the constraint does not belong in the above two cases, the result is uncertain (Class C). Only Class C constraints are explicitly evaluated through the distance computations. Fig. 4 provides examples illustrating the classification algorithm.

Fig. 5 illustrate how classifications are managed. For example, constraint $|I, e| < 2$ is classified as Class A, unsatisfied, because the partition S_{55}, in which object e is located, is an external partition to the region with the static point I as center and 2 as the radius (the dashed line). The distance between e and I must be larger than 2 unless e changes its partition, and thus invokes another constraint classification.

3. ANALYTICAL MODEL

```
procedure Classify_Constraints_NBS(MobileObject p)
/* This function is called when some object p updates its*/
/* partition. It classifies the constraints p associates*/
begin
    for each Constraint c that p is associated
    {
      let P = [p1, p2, ..., pn] be the set of bodies in c;
      if (∃pi ⊂ P, pi ∈ external partition)
        Classify(c, Class_A);
      else if (∀pi ⊂ P, pi ∈ internal partition)
        Classify(c, Class_B);
      else
        Classify(c, Class_C);
    }
end
```

Figure 4: Classification Algorithm (NBS)

In this section, we establish an analytical model for the cost of the constraint evaluation. Through the model, we quantify the matching cost and secondary storage access cost for both algorithms. For the sake of simplicity, we assume that the constraint evaluation is performed in batches. In each batch, all the mobile objects update their location exactly once, and consequently all constraints are evaluated once; so that the constraint solver achieves the highest precision possible with the given location update frequency. In our implementation, the location update is taken as data stream and the evaluation is triggered by the location updates. Table 1 defines the parameters our analytical model is based on.

Table 1: Model Parameters

Parameters	Descriptions
α	the cost for classifying one constraint (into Class A, B or C)
β	the cost for matching one constraint
P_{update}	the probability of an object changing its partition and resulting in a partition update
P_C	the probability that one constraint is classified as Class C
P_{miss}	the probability of a page fault
c	the number of location constraints
n	the number of bodies per constraint
N	the total number of objects, $N \leq cn$

3.1 Cost in Main Memory Environments

In this analysis, we assume all the data is in main memory. Since the SPKDT and grid data structure is constructed once and remain static for a long period of time (we assume the movement pattern of the mobile objects does not change frequently), the cost of maintaining the index data structure is ignored. Therefore, there are two components that contribute to the cost in the constraint evaluation, namely the cost of partition updates of mobile objects and the computation cost for all the constraint evaluations for Class C.

The average number of constraints one object is associated with is nc/N. So the partition update for each object takes on average $nc\alpha/N$ time units; therefore each round, the total cost of the partition updates for all N objects is given by the following equation:

$$C_{Partition\ Updates} = nc\alpha P_{update}. \quad (1)$$

The cost for evaluating Class C constraints per round is given by

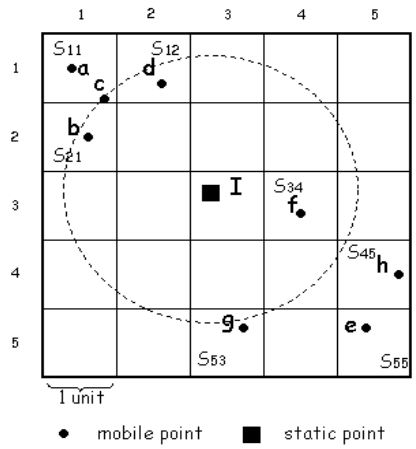

Figure 5: Classification of n-body Static Constraint

the following equation:

$$C_{Evaluation} = c\beta P_C. \quad (2)$$

Thus, the total cost for the NB or NBS algorithm is the sum of Eq. 1 and Eq. 2:

$$C_{total} = nc\alpha P_{update} + c\beta P_C. \quad (3)$$

The cost per round for evaluating constraints with the naïve approach, which evaluates all the constraints sequentially, is simply:

$$C_{Naive} = c\beta. \quad (4)$$

The space partitioning approach outperforms the naïve approach when $C_{total} < C_{Naive}$. This can be achieved when P_{update} or P_C are relatively small.

In the random movement case, P_{update} and P_C is usually very small due to the even distribution of the moving objects and appropriate adjustment of the partition size. For clustered movement of objects, the SPKDT space partitioning can avoid the splitting line going across the clusters and thus greatly reduces the probability of partition updates P_{update}. In the next section, we will experimentally validate these insights.

3.2 Cost under Secondary Storage Access

For applications that need to manage a large number of moving objects with a large number of constraints, secondary storage access cost of the algorithm may become a performance degrading factor. For instance, objects maybe associated with large profiles, as is common in telecommunications applications, or the application may be collocated with other applications, thus not have exclusive access to main memory. The cost (I/O) for accessing data in secondary storage is an order of magnitude larger than accessing data residing in main memory and thus it becomes the major overhead that outweighs anything else. So reducing the number of secondary storage accesses becomes a primary issue.

Both of our algorithms are well suited for reducing I/O access to secondary storage. Note, when secondary storage is accessed, data is fetched in pages rather than in records and each access to secondary storage will transfer a constant unit of data into memory.

Now we consider a simplified case where only the k-d-tree or grid structure is in the memory. User profiles and constraints reside in secondary storage and are loaded to memory on demand. The I/O cost is measured as the number of accesses to the data on the secondary storage device.

For the NB or the NBS algorithm, when some object p_i changes its partition, the algorithm has to access the profiles of the object (1 access), every constraint (i.e., c_j) that p_i is associated with (cn/N accesses) and, in the worst case, all n bodies that c_j is associated with ($< n$ accesses). The cost of a partition update for one object thus is bounded by $1 + cn/N + cn^2/N$. So the total cost for N objects is quantified as: $CIO_{update} < (N + cn + cn^2)P_{update}P_{miss}$. Since all the objects have to be constraint associated, which means $N \leq cn$; therefore

$$CIO_{update} < (2cn + cn^2)P_{update}P_{miss}. \quad (5)$$

For the constraint evaluation, only constraint in Class C and all n objects it relates to are accessed. This part of the cost is given by:

$$CIO_{Evaluation} = (c + cn)P_C P_{miss}. \quad (6)$$

The upper bound of the total I/O cost is the sum of Eq. 5 and Eq. 6:

$$CIO_{total} < ((2cn + cn^2)P_{update} + (c + cn)P_C)P_{miss}. \quad (7)$$

The I/O cost for the naïve approach is simply access number of visits to all the constraints and their associated objects:

$$CIO_{Naive} = (c + cn)P_{miss}. \quad (8)$$

Again with space-partitioning k-d-tree, we are aiming at achieving relatively small P_C and P_{update}, which makes CIO_{total} small.

4. PERFORMANCE ANALYSIS

In this section, we present some of the experimental results that demonstrate the performance of the algorithms. In the experiments we simulate mobile objects moving in a test field of size 40 km $\times$40km. We model two movement patterns, random movement and clustered movement. In the random movement pattern, the Random Direction model is used [26]. All objects are moving with a random speed along a trajectory towards a certain direction; when they reach the boundary of the test field they randomly select a new direction and continue moving. This model maintains a constant density of the objects in the test field. In the clustered pattern, the moving objects form groups and leave the clusters with a small

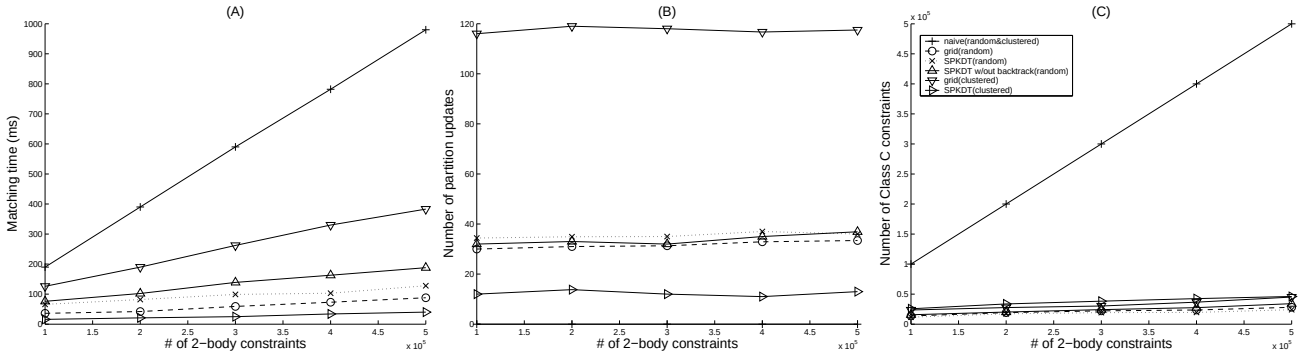

Figure 6: Comparison on Different Indexes

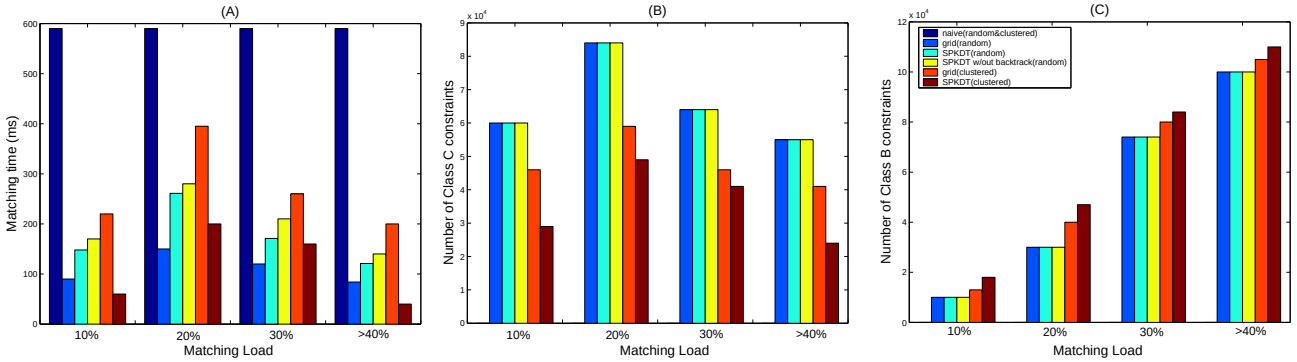

Figure 7: The Effect of Matching Load

probability and the center of the cluster is static. The cluster pattern simulates a city map where some of the spots (shopping malls and stations) represent heavily populated areas. In our experiment, we group the objects into 50 clusters and distribute them evenly across the test field. We set the mean of the velocity of the objects to 5m/s, which simulates walking pedestrians, unless otherwise indicated.

We also compare the grid index with the SPKDT index. Both index schemes partition the whole space into the same number of partitions with roughly the same size. However for the clustered pattern, the splitting lines of the SPKDT are deliberately chosen to avoid crossing the cluster boundaries. Except for Section 4.3, below, we assume the test field is divided into 256 partitions.

Before the experiment, a number of constraints are generated; each constraint is randomly associated with n bodies among a total of 10,000 mobile objects in the field. The alerting distance follows uniform distribution with a certain mean. By changing the mean, the matching load can be adjusted. The matching load is defined as the ratio between the average number of satisfied constraints in each round to the total number of constraints.

For example, for 2-body constraints, with random movement pattern and alerting distance of 400m, 1% matching load is expected on average. However when the alerting distance is 1000m, the matching load increases to 10%. However, in order to obtain a matching load of 1% for the clustered pattern, the alerting distance is set to 100m. Except for Section 4.2, the matching load is exactly 1%.

For simplicity, all moving objects update their location with the same frequency, f_{update} (0.5/sec); we call the time period $1/f_{update}$ seconds one round. In each round, all the moving objects update their location and all the constraints are evaluated exactly once. The choice of the location update frequency only has an effect on the precision of the evaluation. It has no influence on the efficiency of the algorithm evaluation. How to determine the precision for a given update frequency and how to set the appropriate update frequency in order to achieve a desired precision is subject to the future work. Our matching engine is implemented in C++ running on a Pentium IV 2.8GHz. And a detailed description of the system architecture can be found in our previous work [33].

4.1 The Effect of the Number of Constraints

This experiment is intended to evaluate the difference of the performance of various space indexing methods over different movement patterns. Due to the space constraint, we omit the result of NBS, which is very similar to the result of NB. In this experiment, with fixed number of mobile objects (10,000), we keep increasing the number of 2-body constraints and measure the average matching time, the number of partition updates and the number of Class C constraints (Fig. 6). As a baseline, the result of the naïve approach, which evaluates all the constraints sequentially is also shown. In order to compare the number of explicit computations for the constraints, we also plot the number of the constraints evaluated by the naïve approach in Fig. 6(C). Although this is not very appropriate,

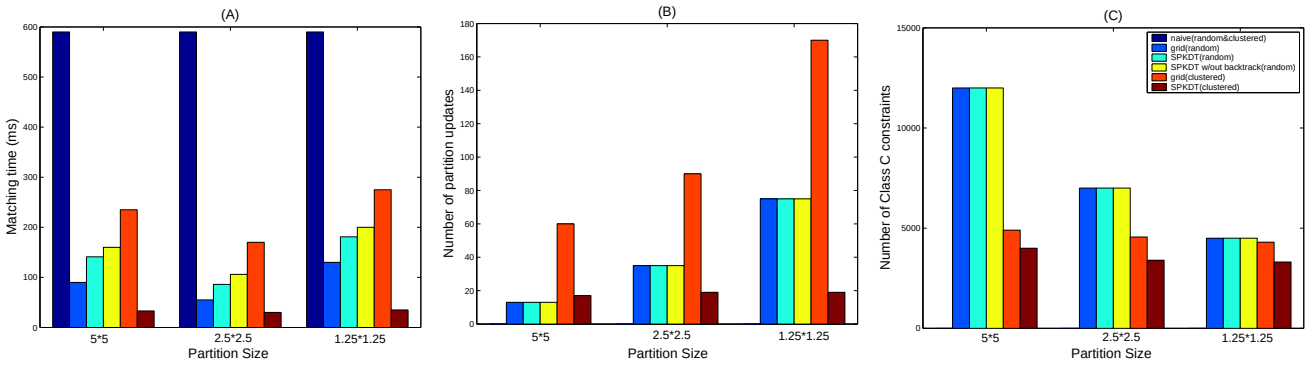

Figure 8: The Effect of the Partition Size

since in the naïve approach there is no classifications.

¿From Fig. 6(A), we observe that for all these index schemes and movement patterns, as the number of constraints increase linearly, the matching time also increases linearly, which complies with the analytical model in Eq. 3. With the random movement pattern, the grid index has slightly lower matching time than the SPKDT due to its efficient computation of the partition updates. However, the SPKDT without backtracking takes more time than SPKDT. With the clustered movement pattern, SPKDT has an average matching time of around 5% of the naïve approach; but the grid index takes about ten times more matching time than the SPKDT. This is because of the overhead of the larger number of partition updates induced by the splitting line crossing the clusters (Fig. 6(B)). For the constraint with more than two bodies, we obtain very similar results.

4.2 The Effect of the Matching Load

This experiment is intended to reveal the effect of the matching load on the performance of the algorithm. Intuitively, as more constraints are satisfied more constraints would have been classified as Class B and C. In the experiment, we alter the mean of the alerting distance to achieve the desired matching load. Fig. 7 shows the matching time for 300,000 2-body constraint over various matching loads ranging from 10% to above 40%.

¿From Fig. 7(A), between the matching load 10% and 20%, the matching time increases along with the matching load for all index schemes and movement patterns. However, when the matching load goes beyond 20%, we observe a decrease of the matching time. Higher matching load, which is induced by larger alerting distances, forces more and more constraints to be classified as Class B (satisfied). The number of Class C constraints starts to drop beyond 20% matching load because at that moment the alerting distance is so large, sometimes it even goes beyond the test field, therefore more constraints become Class B and the number of Class C constraints decreases (Fig. 7(B,C)).

4.3 The Effect of the Partition Size

This experiment evaluates the effect of the partition size on the performance of the algorithm. Intuitively, the smaller the partition, the more partition updates are expected ($P_{update}\uparrow$), however, smaller partitions divide the whole space more finely, the constraint classification becomes more precise. This tends to reduce the number of Class C constraints ($P_C\downarrow$). Therefore the evaluation cost is dropping. There is a tradeoff between the cost of partition update and the evaluation cost. This experiment is intended to quantify this tradeoff.

Fig. 8 shows the matching time, the corresponding number of partition updates and number of Class C constraints for different partition sizes, $5km \times 5km$, $2.5km \times 2.5km$ and $1.25km \times 1.25km$, which divide the whole test field into 64, 256 and 1024 partitions, respectively. Notice, for the SPKDT with clustered pattern, since the splitting lines are deliberately chosen to avoid cutting through clusters, the specified partition size is only roughly complied with. ¿From Fig. 8, we observe the impact of the partition size on the matching time: smaller partition size brings about more partition updates (see Fig. 8(B)), thus more time is required to re-classify the constraints associated with the objects updating their partitions. However, it also leads to less Class C constraints (see Fig. 8(C)). It can be observed that with partition size $2.5km^2$, our algorithm performs best and consumes the least amount of matching time (see Fig. 8(A)). We call this the optimal partition size. Other partition schemes fail because either the cost of partition updates is too high ($1.25km^2$) or the cost of evaluation is too high ($5km^2$). Notice that for the SPKDT with clustered movement pattern, the partition size does not make that much a difference because it already reduces the overhead of partition updates to a minimum. Further study shows that the optimal partition size is also related to the average velocity of the objects. The optimal partition size is almost proportional to the average velocity of the objects. Larger partition size is more in favour of fast speed movement. The NBS algorithm yields similar results.

4.4 Secondary Storage Access Evaluation

In this experiment, we measure the number of disk access required for 10,000 objects and 500,000 2-body constraints. We assume that only the k-d-tree and grid data structure is residing in memory and that everything else must be swapped in and out of memory on demand. As page replacement strategy, we use the least recently used replacement (LRU) scheme. The size of the page is set to be 4096 bytes. To represent one record of a 2-body constraint, we use two 4-byte numbers to identify the objects and one more number for the alerting distance, so the page capacity (the number of records a page can hold) for the 2-body constraint is 341. Similarly, to represent one record of object we need two 4-byte numbers for the 2-d location position information and one 4-byte number for the pointer to each constraint the object is associated with. In our experimental setting, since on average each object is associated with 100 constraints, the average page capacity for objects is 10. Fig. 9

plots the number of secondary storage accesses against the number of pages that are allowed in memory. We observe that the disk accesses of the NB algorithm is much less than that of the naïve approach because of the limited number of records accessed, due to the pruning capability of the partitioning scheme. In the random movement case, the SPKDT has very similar secondary storage access cost as the grid index in terms of the user profiles (objects) and constraints accessed. In the clustered pattern, the SPKDT performs much better than the grid index because it trims away the unnecessary partition updates. The NBS algorithm exhibits similar behavior.

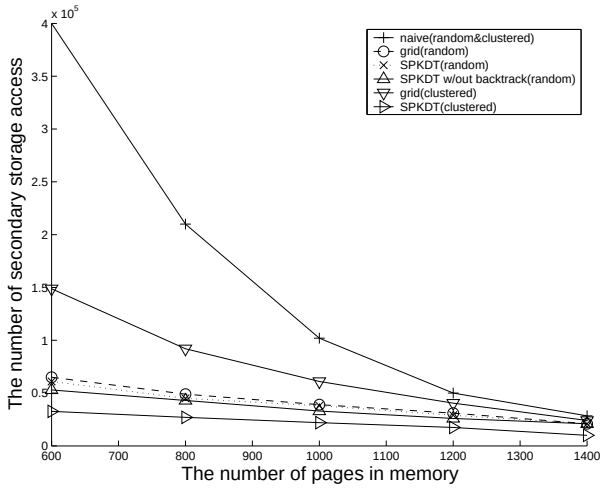

Figure 9: Secondary Storage Access

5. SYSTEM IMPLEMENTATION

This section briefly describes an overall system architecture and implementation that demonstrates an application of the developed location constraint matching algorithms in this paper. This architecture has been demonstrated in a simulated environment in Xu and Jacobsen [33]. The system is currently deployed in a real cellular network environment to support services based on location position correlation for mobile subscribers. A companion paper describes the current status and provides further details about this architecture and system implementation [34].

Our research prototype is referred to as L-ToPSS (Location-based Toronto Publish/Subscribe System), which is part of the extended ToPSS family (Toronto Publish/Subscribe System [18, 17, 9, 25, 8, 24].) L-ToPSS aims to support location-based services in a push-oriented style [8, 33, 34].

Publish/subscribe systems enable their clients to exchange information by publishing events (i.e., publishing clients) and subscribing to events of interest (i.e., subscribing clients). The publications and subscriptions are maintained, processed, and evaluated in a broker infrastructure [10]. This infrastructure can be either centralized or distributed. Publishers and subscribers are decoupled in location, space, time, and representation.

The location constraint matching problem defined in this paper can be looked at as an instance of the publish/subscribe matching problem [10]. Static or dynamic n-body constraints are subscriptions. Location updates are publications. The problem is to determine for a given set of subscriptions (i.e., constraints) and a given publication (i.e., update(s) of location information of mobile entities), which of the registered subscriptions match the given publication.

The L-ToPSS system architecture is shown in Fig. 10. The figure shows the architecture at a high level comprising an input staging area that processes location constraints and location updates entering from the outside, the constraint solver implementing the algorithms described in this paper, and the notification engine communicating matches back to subscribers. Constraints are submitted to the system, either through a web-server (i.e., entered from subscribers through a web site) or directly from the mobile subscriber's device over a data link to the L-ToPSS server. Changing location updates trigger the evaluation of subscriptions stored in the system. The constraint solver stores all the location constraints (both n-body and n-body static constraints) based on the data structures outlined in this paper.

In terms of the L-ToPSS infrastructure, mobile subscribers are distinguished by a unique Mobile Identification Number (MIN). The location information is represented as a (`$MIN_i$, time, current_longitude, current_latitude, current_altitude`) tuple. This tuple is forwarded to the constraint solver to trigger the constraint evaluations.

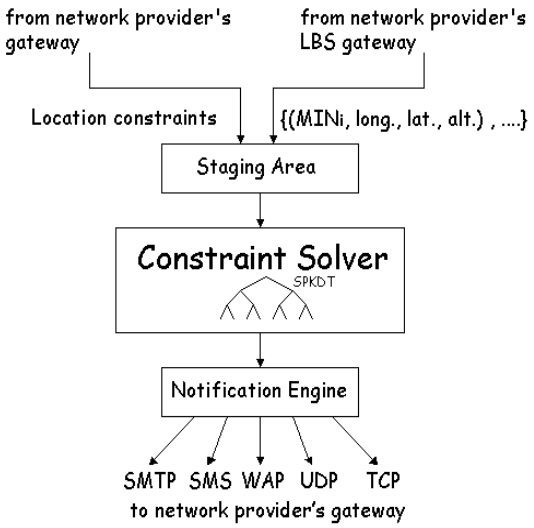

Figure 10: Overall L-ToPSS System Architecture

In a stand-alone implementation of our prototype, we use two tools to facilitate the demonstration and the evaluation of L-ToPSS. These tools are the City Simulator [13] and the Location Transponder [20]. The City Simulator generates spatial data traces simulating the movement of mobile subscribers in a city. Up to a million subscribers can be simulated with this tool. The Location Transponder is used to schedule the generated movement traces as real-time input to the L-ToPSS matching infrastructure. Each location update triggers the evaluation of the stored location constraints. A satisfied constraint will yield a notification as specified for the location constraint. For example, all objects involved in the constraint, one designated object involved in the constraint, or an outside, unrelated object may receive the notification signaling the satisfaction of the given constraint. For more detailed information about an extension of the L-ToPSS prototype operating in a cellular network refer to Xu and

Jacobsen [34].

6. RELATED WORK

Spatial indexing methods have been widely studied in recent years [12]. There are mainly two classes of spatial indexing, point access methods (PAM) and spatial access methods (SAM). These methods provide support for efficient access of spatial objects in database. PAM are used for the search of a set of points in space (point query); the typical methods include the grid file [21], hB-tree [19] and k-d-tree [5]. SAM support access of objects with spatial extension. Recently, R-tree based indexes [14, 28, 27] have been used as a spatial access method for moving objects-based queries. Other moving object indexes include Time-Oblivious indexing [3], dynamic external memory data structures [16] and velocity constrained indexing [29]. These techniques can be used for indexing moving objects in space to efficiently evaluate a query over the moving objects, such as the future point query and the range query [29, 6]. These indexing techniques aim at pruning the search space so that only a limited number of objects are visited to enhance the search speed. However, no explicit support is provided for tracking and correlating constraints between objects (e.g., the distance between objects, for example), which is crucial in our context. Many of the indexing approaches on moving objects are based on the assumption that all the objects are moving along a linear trajectory with the same speed or with a speed in a given range [16, 29], which is not very realistic for the scenarios we are targeting.

Kinetic data structures [4], which maintain attributes of interest to manage, evaluate, and query the motion of geometric objects do not directly apply for solving the location constraint matching problem as stated in this paper. Our interest is with processing large numbers of constraints over the movement of many moving objects.

The buddy tracking system [30] is the only work known to us that is looking at a problem statement similar to the location constraint matching problem we are stating. They are, however, exclusively looking at a 2-body problem in the 2-d space and propose and evaluate a distributed algorithm for solving this problem. It is assumed that the mobile objects communicate with each other directly to resolve the 2-body constraints. Their objective is to reduce the communication cost. They also sketch a quadtree-based algorithm. However, their quadtree-based approach only solves 2-body constraints and all the partitions have to be of the same size; so a skewed clustered movement pattern could only be supported with difficulty. Moreover, their solution is restricted to one global alerting distance for all registered constraints, which we consider a severe limitation for the targeted problem context. The buddy tracking approach also lacks an evaluation of the quadtree-based algorithm.

Constraint databases (CDB) aim at representing large or infinite sets in compact ways [11]. A constraint can be a linear or polynomial equation. CDB is being applied to the modeling and integrating of spatial and temporal data [11]. The research in CDB has been focusing on the fundamental issues, like data modeling [23] and query language definitions [7]. We are not aware of any work on using CDB for evaluation large number of constraints evaluated over streams of continuously changing location information, such as the location constraints we are mostly interested in.

The location constraint matching problem also reminds one of the publish/subscribe matching problem [10]. Indeed, location constraints could be interpreted as subscriptions and location updates as publications. However, location constraints are not of the form commonly assumed by publish/subscribe systems, which makes the application of these matching algorithms difficult. An extension of publish/subscribe for processing location-aware data (i.e., notify a subscriber with a matching subscription close to a publishing entity) has been proposed by Burcea and Jacobsen [8].

7. CONCLUSION AND FUTURE WORK

Location constraint processing is essential for applications, such as location-based services that aim at tracking, correlating, and filtering information about moving entities. In this paper, we define two types of location constraints, the n-body constraint and the n-body static constraint, which capture a *close-to-relation* among sets of moving entities. We propose the NB and NBS algorithms to evaluate large sets of these two constraints, respectively. Our approach is based on the space-partitioning k-d-tree and, an alternative is based on the grid index, which partitions the whole space into smaller parts. The distance between the partitions serves as a rough measure for the bound on the distance between objects lying inside these partitions. Using the partition information of the moving objects, only the constraints that are likely satisfied (yet uncertain) are chosen for further consideration. Our experimental results show that with a random movement pattern, the grid index is slightly better than the SPKDT index due to a more efficient partition computation; however for the clustered pattern the SPKDT index becomes much better than the grid index because it reduces the cost of the partition updates. We also find that the matching time reaches its upper bound when the matching load is above 20%. We further show that our approach is well suited for large constraint loads that go beyond capabilities of main memory processing environments. Such loads are expected from emerging location-based services in wireless networks and for other object tracking and correlation applications.

In our future work, we intend to model the correlation between the certainty of a constraint match and the projected object positions at time of match and time of notification. That is, besides reporting the constraints that are satisfied, we will also report on the likely accuracy of the satisfied constraint at a given point in time. This will provide insights on how to adjust the location update frequency in order to achieve a desired level of accuracy. We also intend to study how to predict possible future matches and and how to perform matching in environments with obstacles.

8. REFERENCES

[1] New and enhanced features of fedex insight. *http://www.fedex.com/us/*.

[2] Radio frequency identification systems (RFID). *http://www.ti.com/tiris/docs/solutions/animal/livestock.shtml*.

[3] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Symposium on Principles of Database Systems*, pages 175–186, 2000.

[4] Basch, Guibas, and Hershberger. Data structures for mobile data. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM.*, pages 18:509–517, 1975.

[6] Mark De Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry. Springer-Verlag New York, 2000.

[7] Alexander Brodsky and Yoram Kornatzky. The lyric language: Querying constraint objects. In *SIGMOD Conference*, pages 35–46, 1995.

[8] Ioana Burcea and Hans-Arno Jacobsen. L-ToPSS - Push-oriented Location-based Services. In *4th VLDB Workshop on Technologies for E-Services (TES'03)*, 2003.

[9] Ioana Burcea, Hans-Arno Jacobsen, Eyal de Lara, Vinod Muthusamy, and Milenko Petrovic. Disconnected Operation in Publish/Subscribe Middleware. *2004 IEEE International Conference on Mobile Data Management*.

[10] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD*, 30(2):115–126, 2001.

[11] Leonid Libkin Gabriel Kuper and Jan Paredaens. Constraint databases. Springer Verlag, 2000.

[12] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[13] James Kaufman Jussi Myllymaki and Jared Jackson. City Simulator. IBM alphaworks emerging technologies toolkit, http://www.alphaworks.ibm.com/tech/citysimulator, November 2001.

[14] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 500–509, Santiago, Chile, 1994.

[15] David R. Karger. Finding nearest neighbors in growth-restricted metrics. In *In Proc. ACM Symposium on Theory of Computing (STOC '02)*, 2002.

[16] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. In *Proceedings of the 18th ACM PODS Conference, May 31 - June 2, 1999,Philadelphia,Pennsylvania*, pages 261–272. ACM Press, 1999.

[17] Gouli Li, Shuang Hou, and Hans-Arno Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems based on Modified Binary Decision Diagrams. *International Conference on Distributed Computing Systems (ICDCS'05)*.

[18] Haifeng Liu and H-Arno Jacobsen. Modeling uncertainties in Publish/Subscribe System. In *In Proceedings of ICDE*, 2004.

[19] David B. Lomet and Betty Salzberg. A robust multiattribute search structure. In *Proc. 5th ICDE*, pages 296–304, 1989.

[20] Jussi Myllymaki and James Kaufman. Location Transponder. IBM alphaworks emerging technologies toolkit, http://www.alphaworks.ibm.com/tech/transponder, April 2002.

[21] Hinterberger H. Nievergelt, J. and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In *ACM Trans. Database systems 9, 1984*.

[22] Jeff Erickson Pankaj K. Agarwal, Lars Arge. Indexing moving points. In *In Proc. 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

[23] Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. Towards a theory of spatial database queries (extended abstract). In *Proc. ACM PODS*, 1994.

[24] Milenko Petrovic, Ioana Burcea, and H.-Arno Jacobsen. S-ToPSS - a semantic publish/subscribe system. In *Very Large Databases (VLDB'03)*, Berlin, Germany, September 2003.

[25] Milenko Petrovic, Haifeng Liu, and Hans-Arno Jacobsen. G-ToPSS - fast filtering of graph-based metadata. In *the 14th International World Wide Web Conference (WWW2005*, Chiba, Japan, May 2005.

[26] Elizabeth M. Royer, P.Michael Melliar-Smith, and Louise E. Moser. An analysis of the optimum node density for ad hoc mobile networks. In *Proceedings of the IEEE International Conference on Communications*, pages 857–861, June 2001.

[27] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD, Dallas, Texas, USA*.

[28] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R-Tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.

[29] Dimitri Kalashnikov Walid Aref Susanne E. Hambrusch Sunil Prabhakar, Yuni Xia. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, October 2002.

[30] Arnon Amir. Alon Efrat. Jussi Myllymaki. Lingeshwaran Palaniappan. Kevin Wampler. Buddy tracking - efficient proximity detection among mobile friends. In *Infocom 2004*.

[31] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The active badge location system. Technical Report 92.1, Olivetti Research Ltd. (ORL), 24a Trumpington Street, Cambridge CB2 1QA, 1992.

[32] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, LNCS. Springer, 1991.

[33] Zhengdao Xu and H. Arno Jacobsen. Efficient constraint processing for highly personalized location based services. In *30th International Conference on Very Large Data Bases (VLDB04) Toronto, Canada*, 2004.

[34] Zhengdao Xu and H. Arno Jacobsen. Demo: A framework for location information processing. In *6th International Conference on Mobile Data Management (MDM'05), Ayia Napa, Cyprus*, 2005.

[35] Jun Zhang, Manli Zhu, Dimitris Papadias, Yufei Tao, and Dik Lun Lee. Location-based spatial queries. *In SIGMOD Conference*, 2003.

[36] Yilin Zhao. Standardization of mobile phone positioning for 3G systems. *IEEE Communication Magazine, July 2002*.