Adaptive Location Constraint Processing

Zhengdao Xu and Hans-Arno Jacobsen
University of Toronto
zhengdao@cs.toronto.edu, jacobsen@eecg.toronto.edu

ABSTRACT

An important problem for many location-based applications is the continuous evaluation of proximity relations among moving objects. These relations express whether a given set of objects is in a spatial constellation or in a spatial constellation relative to a given point of demarcation in the environment. We represent proximity relations as location constraints, which resemble standing queries over continuously changing location position information. The challenge lies in the continuous processing of large numbers of location constraints as the location of objects and the constraint load change. In this paper, we propose an adaptive location constraint indexing approach which adapts as the constraint load and movement pattern of the objects change. The approach takes correlations between constraints into account to further reduce processing time. We also introduce a new location update policy that detects constraint matches with fewer location update requests. Our approach stabilizes system performance, avoids oscillation, reduces constraint matching time by 70% for in-memory processing, and reduces secondary storage accesses by 80% for I/O-incurring environments.

Categories and Subject Descriptors

H. [**Information Systems**]; H.2.8 [**Database Applications**]: Spatial databases and GIS; H.3.3 [**Information Search and Retrieval**]: Information Filtering; H.4 [**Information Systems Applications**]: Location-based Services

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Location-based Services, Location Constraint Processing, Moving Object Indexing, Adaptive Indexing, Location Update Policy, Location Query, Standing Query, Continuous Location Query, and Constraint Matching.

1. INTRODUCTION

The pervasive presence of wireless networks combined with advances in location positioning technology [28] gives rise to new and previously unthinkable potentials for tracking, correlating, and filtering of information about moving objects. For example, it is possible to track the location of mobile users in a wireless network, even across wireless coverage areas, and in buildings across campus [28, 19], to track the discrete location of vehicles or packages in delivery [1], and even track 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*, *no closer than*, or *further-away-than* a specified distance to each other or to a given point of demarcation in the environment. We refer to this problem as the *location constraint matching problem*. A given set of objects specifying a proximity relation that must be enforced is referred to as a *location constraint*. There are many applications that require a solution to this problem. The challenge is to device algorithms that can efficiently monitor large numbers of location constraints over larger numbers of continuously moving objects.

In a cellular 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 (e.g., friend finder and buddy tracking applications) or are *close to* a designated point in the environment (e.g., the CN Tower in Toronto or a road congestion on Highway 407.) In metropolitan areas the number of objects tracked can easily reach hundreds of thousands, specifying a similar number of location constraints.

Other examples draw from protecting security sensitive areas in cities by specifying that no airplane is to come *closer than* a specified distance from designated points, such as nuclear plants, towers, stadiums etcetera. Air traffic controllers and pilots may benefit from alerting services that warn if designated planes come *too close* to each other. Similarly, major production plants may specify policies that restrict personnel from entering restricted areas or enforce constraints that a specific area must be guarded by a set number of personnel. Again, these scenarios may involve large numbers of continuously moving objects with many associated location constraints. Altogether different applications can be found in multi-player online games, where friend or foe may want to setup constraints to notify and warn of each others' presence and proximity in the game world or parts of the virtual space. Multi-player online games are played by thousands of players at the same time. The efficient tracking of constraints among players and among players and the game world is of critical importance to the success of the game.

To model these scenarios and to express various types of proximity relations, we introduce two classes of location constraints, the n*-body constraint* and the n*-body static constraint*. We refer

SIGMOD'07, June 11–14, 2007, Beijing, China.

to these constraints as *location constraints*, as they are evaluated over the location data of the associated objects. The constraints are defined as follows:

The *n-body constraint* is of the form $|{p_1}^t, {p_2}^t, ..., {p_n}^t| < (or >) d$, and also abbreviated as $P < (or >) d$ for object set P(=$\{p_1, p_2, ..., p_n\}$). The constraint with operator $<$ is *satisfied* if the n moving objects, identified by, $p_1, p_2, ..., p_n$, can be enclosed by a circle with a diameter smaller than d at time t. The constraint with operator $>$ is *satisfied* if the diameter of the smallest circle enclosing the objects is greater than d at 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*.

The *n-body static constraint* is of the form $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t| < (or >) d$, also abbreviated as $|A, P| < (or >) d$ for object set P (=$\{p_1, p_2, ..., p_n\}$). A is the coordinate of some static point. The constraint is *satisfied* if the n moving objects, identified by, $p_1, p_2, ... , p_n$, are within (for operator $<$) or outside (for operator $>$) the circle defined by d around the static point A at time t.

The above constraint definitions do not include $=$, $\leq$ and $\geq$, as the resulting constraints can be easily expressed with the negation of the constraint itself. For example, $P_{c_i} \leq d$ is simply $\neg(P_{c_i} > d)$ and $P_{c_i} = d$ is equivalent to $\neg(P_{c_i} > d) \wedge \neg(P_{c_i} < d)$, where P_{c_i} denotes the moving objects associated with constraint c_i. This simplifies the design of the constraint matcher.

The location constraint matching problem can then be formally stated as follows: Given a set of location constraints, $C = \{c_1, c_2, ..., c_k\}$, which designate the 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*.

Location constraints are like continuous queries that once submitted to the system remain active until explicitly revoked. Fig. 1 shows the data flow of a typical location-based service employing location constraint matching, as defined in this paper. Location updates of mobile objects are streamed into the system and trigger the evaluation of constraints and the constraint satisfaction is communicated back to the interested subscribers via notifications.

Existing data management and indexing techniques for moving

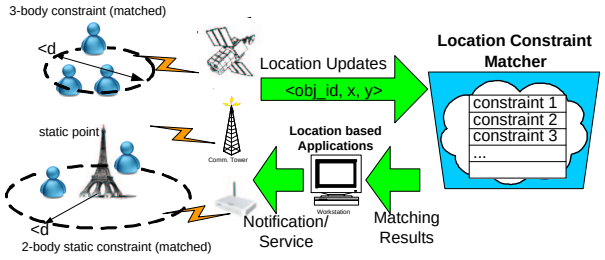

Figure 1: Data Flow for Location Constraint Processing

objects [11, 3, 16, 17, 14] are mainly focusing on the efficient processing of a single spatio-temporal query, such as a range query and the k nearest neighbor query (kNN). However, the location constraint matching problem deals with the concurrent and continuous evaluation of many constraints at the same time (i.e., hundreds of thousand of location constraints for the applications we advocate.) Due to the continuous and frequent movement of objects, an underlying database would be mostly busy updating the objects' location without being able to evaluate the constraints. Moreover, the location constraint matching problem is different from the kNN problem [10, 27] in that the kNN problem determines the k nearest object(s) among all the objects in the space to a given point, while the location constraint matching problem continuously monitors whether a specific set of objects are in a given spatial constellation to each other or to a given point of demarcation in the environment.

In this paper we propose an adaptive space partitioning scheme and algorithms to solve the location constraint matching problem. We experiment with two space partitioning schemes, one is a tree-based organization of the space and the other one a grid file-based organization. Our solution supports the continuous tracking of large numbers of location constraints for large populations of moving objects. Our algorithms are designed to adapt to changes in location constraint loads and changes in movement patterns of objects. In particular, our solution adapts to clusters of objects changing over time in size, skewness and the formation of new clusters. Furthermore, our solution constitutes a framework customizable for a wide-range of query types expressing different geometric relations among moving objects. We demonstrate and evaluate alternative query types such as the *continuous reverse range query* and the *continuous reverse k nearest neighbor query*. We also introduce the notion of *conflicting* and *harmonious* constraints to capture dependency relations among constraints that help in constraint pruning. We develop a detailed analytical model to assess the constraint evaluation cost for in-memory and I/O-incurring environments. Finally, we experimentally validate the efficiency of our algorithms based on various movement patterns, compare to alternative approaches, and demonstrate scalability.

In Section 2, we describe the adaptive location constraint matching algorithm. In Section 3, we develop the analytical model to assess the constraint evaluation cost, determine system parameters, and estimate secondary storage access cost. The system architecture of a fully implemented location constraint processing system is reviewed in Section 4. Section 5 presents the experimental evaluation of our algorithms. In Section 6, we put our work in perspective to related approaches.

2. MATCHING ALGORITHM

Our solution to the location constraint matching problem is based on space partitioning. The intuition is that space partitioning can approximate the location of the moving objects, without the need for the exact location of objects all the time, to resolve many constraints with certainty. In this section we describe our approach in detail.

2.1 Background & Basic Algorithms

A solution to the location constraint matching problem in prior work [23] divides the whole space into partitions that are indexed with a k-d-tree [5] or a multi-layered grid file structure [13]. To avoid the overhead of a top-down scan in the k-d-tree when objects change partition, a backtracking algorithm is used to quickly associate the object with the new partition. In a multi-layer grid file, the space is partitioned with different layers of grids, the grids in the same level are equal-sized cells which are further indexed with hashtables.

We assume that the position of each object is retrieved with GPS, ground-based sensors or other location positioning technology. We call each position retrieval a *location update* and we call the movement of an object from one partition to another a *partition update*.

Constraint evaluation consists of two parts, the constraint evaluation based on exact position information and the partition-based constraint evaluation given partition information. For the evaluation based on the exact position, Welzl's algorithm [20] is adopted, which computes the smallest circle that encloses n points in $O(n)$ time. Below, we introduce partition-based evaluation that is triggered when an object updates its partition or when the partition scheme is adjusted by the adaptive adjustment algorithm. The par-

tition information is used as a rough approximation for the position of the moving objects and for many constraints, this approximation is enough to determine a result.

Partition-Based Evaluation for n-body Constraints: For n-body constraint, if some object p_i moves into a new partition S_i, all the constraints it is associated with need to be re-evaluated based on the partition information. Suppose that c_j is the constraint that p_i is associated with and $P_{c_j} = \{p_1, p_2, ..., p_n\}$ is the set of bodies involved in c_j ($p_i \in P_{c_j}$). After the partitions that contain objects in P_{c_j} are identified, the smallest circle enclosing all these partitions and the smallest circle intersecting all these partitions are computed. The size of these two circles are the upper and lower bound of the size of the actual circle that encloses the objects in P_{c_j}. Based on this upper and lower bound, the result of many constraint can be determined.

For instance, the Partition_Based_NB Algorithm for the constraints with the "$<$" operator ($|{p_1}^t, {p_2}^t, ..., {p_n}^t| < d$) is shown below. If the diameter of the smallest intersecting circle (lower bound) is larger than d (checked by function `Intersect` in Line 3), c_j is *unsatisfied*. If the smallest enclosing circle (upper bound) has a diameter smaller than d (checked by function `Enclose` in Line 5), c_j is *satisfied*. If d is between this lower and upper bound, the constraint is *uncertain* (Line 8). Due to the symmetry, the constraint $|{p_1}^t, {p_2}^t, ..., {p_n}^t| > d$ can be solved by exchanging Line 4 with Line 6.

The result of an evaluation is valid as long as the objects remain in their partitions and the partition scheme does not change. Fig. 2 illustrates partition-based evaluation. The 3-body constraint $c_1(|p_1, p_2, p_3| < 3$, right side of the figure) is *satisfied* because the diameter of the circle O_1 which encloses partitions S_{11}, S_{21} and S_{22} [1] (containing p_1,p_2 and p_3) is $\sqrt{8}$ (< 3). The constraint c_3 is *unsatisfied* because the diameter of the circle O_2 which intersects S_{21}, S_{22} and S_{45} is $\sqrt{10}$ (> 3). However, the constraint c_5 is *uncertain* because the diameter of the circle O_3 enclosing S_{36}, S_{45} and S_{56} is $\sqrt{10}$ and the diameter of the circle O_4 intersecting these partitions is 1, but the alerting distance 3 is between the lower and upper bound ($\sqrt{10} > 3 > 1$). Further computation based on the precise object position reveals that c_5 is actually *satisfied* (circle not shown for clarity). But this can not be established with partition information alone. Similarly, one can verify the evaluation of constraints c_2, c_4 and c_6.

Algorithm Partition_Based_NB($MobileObject\ p$)
1. **for** each Constraint c that p is associated with
2. **let** $P = \{p_1, p_2, ..., p_n\}$ be the set of bodies in c;
3. **if** (`Intersect`($\bigcup_{i=1}^{n} p_i.current\_partition$)$> c.d$))
4. $c.result$ = *unsatisfied*;
5. **else if** (`Enclose`($\bigcup_{i=1}^{n} p_i.current\_partition$)$< c.d$)
6. $c.result$ = *satisfied*;
7. **else**
8. $c.result$ = *uncertain*;

Partition-Based Evaluation for n-body Static Constraints: The n-body static constraint, $|A, {p_1}^t, {p_2}^t, ..., {p_n}^t| < d$, is matched if and only if all p_i ($i \in 1..n$) are in the circle O with static point A as center and d as radius. Depending on whether the partition is inside, intersecting, or outside the boundary of O, we identify the *internal*, *bounding*, and *external* partitions w.r.t. O (e.g., the partition completely inside O is defined as an *internal* partition and so on.) The distance between the moving object and the static point A can be tracked according to the type of the partition the object is in (i.e., "*internal*", "*external*", or "*bounding*").

The partition-based evaluation (Partition_Based_NBS) for constraints with the "$<$" operator works as follows: if some object in $P = \{p_1, p_2, ..., p_n\}$ is in the *external* partition, the constraint is *unsatisfied* (Line 4); if all objects are in the *internal* partition, the constraint is *satisfied* (Line 6); if the constraint does not belong to the above two cases, the constraint is *uncertain* (Line 8). The solution to the constraint with the "$>$" operator can be derived in a similar fashion due to its symmetry with "$<$".

[1] S_{ij} represents the partition in the ith row and jth column

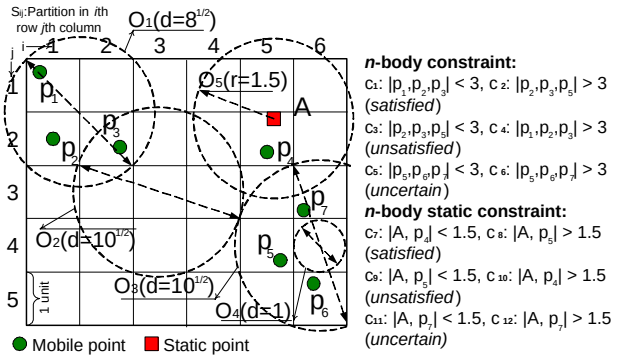

Figure 2: Illustration of Partition-based Evaluation

In Fig. 2, the 1-body static constraint c_7 and c_8 are *satisfied* because w.r.t. circle O_5, partition S_{25} (contains p_4) is an *internal* partition and S_{45} (contains p_5) is an *external* partition. Similarly, we can verified that c_9 and c_{10} are *unsatisfied*. However, constraints c_{11} and c_{12} are *uncertain* because S_{36} (contains p_7) is a *bounding* partitions w.r.t. O_5.

Algorithm Partition_Based_NBS($MobileObject\ p$)
1. **for** each Constraint c that p is associated
2. **let** $P = \{p_1, p_2, ..., p_n\}$ be the set of bodies in c;
3. **if** ($\exists p_i \in P, p_i \in$ *external* partition)
4. $c.result$ = *unsatisfied*;
5. **else if** ($\forall p_i \in P, p_i \in$ *internal* partition)
6. $c.result$ = *satisfied*;
7. **else** // in *bounding* partition
8. $c.result$ = *uncertain*;

Constraint Evaluation Algorithm: A location update triggers constraint evaluation (i.e., Evaluation). We distinguish the cases where the location change leads to a partition update and where no such partition update is incurred.

For a partition update, all constraints an object is associated with have to be evaluated using the partition-based evaluation function (Lines 4, 5) and the *uncertain* constraints involving that object have to be explicitly evaluated based on the exact location position information afterwards (with `Match_Uncertain_Constraints` in Line 6). The number of *uncertain* constraints (expressed as $U_{parti\ tion\_id}$) in the previous and current partitions are updated afterwards (Line 7). On the other hand, if the location update does not incur a partition update, only *uncertain* constraints need to be evaluated (Line 9). Partition-based evaluation is invoked every time an object changes a partition or when the partition is adjusted, but not for every location update.

Algorithm Evaluation($MobileObject\ p$)
1. $p.previous\_partition = p.current\_partition$;
2. $p.current\_partition$ = `Find_New_Partition`(p);
3. **if** $p.previous\_partition \neq p.current\_partition$
4. `Partition_Based_NB`(p);
5. `Partition_Based_NBS`(p);
6. `Match_Uncertain_Constraints`(p);
7. update $U_{p.current\_partition}$ and $U_{p.previous\_partition}$;
8. **else** *if $p.previous\_partition = p.current\_partition$*
9. `Match_Uncertain_Constraints`(p);

2.2 Adaptive Space Partitioning

The static space partitioning described above suffers in environments where both the constraints and the movement patterns of the

objects are continuously changing. For many applications, constraints are continuously updated based on changing user preferences (i.e., inserts and deletes.) Thus, a partition scheme optimized for one set of constraints and movement pattern may not be optimal for another set of constraints and a different movement pattern. Also, partition updates incur overhead when a large number of objects are moving between partitions, which is to be expected for applications with varying movement patterns. These characteristics are the motivations to develop an adaptive space partitioning scheme that evolves with change.

The AKDT and AMLG Index: We propose the adaptive k-d-tree (AKDT) index and the adaptive multi-layer grid (AMLG) index to accommodate this situation. AKDT is fundamentally a k-d-tree, except that it partitions the space rather than the objects in the space. In AKDT, each leaf node represents a single partition in the space. Each internal node stores information about a splitting line and represents the union of partitions of the (leaf) nodes underneath it. Similarly, in AMLG, each grid at the lowest layer presents a single partition in space and the higher layer grid stores the information of the grid layout and represents the union of all partitions underneath this layer. AKDT and AMLG adaptively set and relinquish the splitting lines.

The adaptive space partition algorithm consists of two stages, namely, the *initial partitioning* stage and the *adjustment* stage. The *initial partitioning* simply divides the whole space into small cells with arbitrary granularity. The adjustment stage tunes the partition scheme with an adjustment frequency proportional to the linear combination of the average velocity over all moving objects tracked, $\overline{v}$, and the average rate of constraint updates, $\overline{r}$, i.e., $a_1\overline{v} + a_2\overline{r}$, for constants a_1 and a_2. The average velocity is derived from the location updates recorded over time. In this way, the partition scheme evolves as the movement pattern and constraint load change. We formally show with Lemma 2 in Section 3 that the splitting of a partition can reduce the number of *uncertain* constraints and can increase the partition update rate, while the merging of partitions can reduce the partition update rate and can increase the number of $uncertain$ constraints. The objective is to balance the trade-off between splitting and merging such that both the number of *uncertain* constraints and the partition update rate decrease to the greatest extent possible.

For the efficient adjustment, each partition S_i (i is the partition ID) maintains the following additional information: 1. PUR_{S_i}, the estimated partition update rate inside S_i. It equals $\rho_{S_i} * \overline{v}_{S_i} * L_{S_i}$, where ρ_{S_i}, $\overline{v}_{S_i}$ and L_{S_i} are the object density, the average velocity and the total length of the splitting lines inside S_i, respectively. 2. U_{S_i}: The number of *uncertain* constraints associated with the objects in partition S_i. The adjustment process is composed of the split Split_Adjust Algorithm and the merge Merge_Adjust Algorithm. The split algorithm creates new smaller partitions from a leaf node partition by adding tentative splitting lines (Line 2) inside the candidate leaf node. For AKDT, the tentative splitting line generated in the split process is random in terms of its orientation (horizontal or vertical) and its position is strict (e.g., bisects the partition) or follows some given distribution (e.g., uniform or normal distribution). The randomization of the splitting lines reduces the potential for oscillation of the algorithm. For AMLG, the splitting lines do not have this degree of freedom, they have to divide the space into equal-sized partitions. However, the objects in each partition are in the same grid file indexed by a hashtable. The split process reduces the number of *uncertain* constraints, but it also induces more partition update overhead. We evaluate this trade-off later on.

The algorithm only commits the splitting lines in the partition where they lead to the greatest reduction in the number of *uncertain* constraints (recorded by set Γ_U in Line 4) and the smallest increase in partition update rate (recorded in Γ_{PUR} in Line 5). Only the tentative splitting lines inside partitions $\Gamma_U \cap \Gamma_{PUR}$ are confirmed (Line 6). The final number of partitions split depends on the size of Γ_U and Γ_{PUR} that are controlled by the window size w, which is the main indicator of the adaptation speed and is directly coupled with, and therefore adjusted according to, the linear combination of the average speed of movement and the constraint update rate, $a_1\overline{v} + a_2\overline{r}$. The reduction in the number of *uncertain* constraints is computed in Line 3. Fig. 3 illustrates the split adjustment.

Algorithm Split_Adjust(**$int\ w$**)
1. **for** each candidate leaf node S_i
2. $S_i^{'} = S_i$ split by tentative splitting lines;
3. $\triangle U_{S_i}$ = reduction of *uncertain* constraint fr. S_i to $S_i^{'}$;
4. Γ_U = top w partitions with highest $\triangle U_{S_i}$;
5. Γ_{PUR} = top w partitions with lowest PUR_{S_i};
6. replace S_i with $S_i^{'}$ if $S_i \in \Gamma_U \cap \Gamma_{PUR}$;

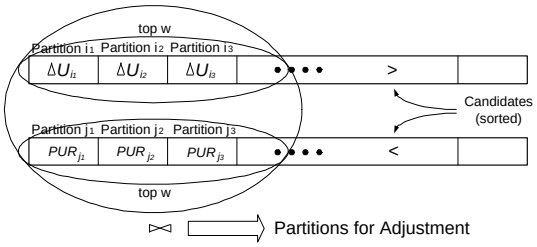

Figure 3: Split Adjustment

The Merge_Adjust Algorithm performs an inverse operation to the split algorithm. It removes the current splitting lines and merges the leaf node partitions. The merge algorithm aims at reducing the partition update rate at the cost of a minor increase in the number of *uncertain* constraints.

Algorithm Merge_Adjust(**$int\ w$**)
1. **for** each candidate second level node S_i
2. $S_i^{'} = S_i$ with splitting lines tentatively removed;
3. $\triangle U_{S_i}$ = increase of *uncertain* constraint fr. S_i to $S_i^{'}$;
4. Γ_U = top w partitions with lowest $\triangle U_{S_i}$;
5. Γ_{PUR} = top w partitions with highest PUR_{S_i};
6. replace S_i with $S_i^{'}$ if $S_i \in \Gamma_U \cap \Gamma_{PUR}$;

Overhead Reduction for Split & Merge Counting the change in the number of *uncertain* constraints by recomputing all the constraints associated with the partition (Line 3 in Split_Adjust and Merge_Adjust) is too costly. Lemma 2 in Section 3 shows that the *uncertain* constraints after the split are a subset of the *uncertain* constraints before splitting, therefore, only the original *uncertain* constraints are considered when computing the number of *uncertain* constraints after splitting. A similar property holds for merging. This reduces the overhead of the split and merge process, however, there are better ways to further reduce this cost.

An *uncertain* constraint may be associated with multiple objects in different partitions and adding splitting lines into these partitions may have different effects on this constraint. For example, in Fig. 4, c_1 ($|p_1, p_2, p_3| < 6$) is associated with partitions S_1, S_2 and S_4 (where p_1, p_2 and p_3 are located). c_1 is *uncertain* because the circle, O_1, enclosing these partitions has a diameter $\sqrt{52}$ (> 6) and the circle, O_2, intersecting these partitions has a diameter $\sqrt{10}$ (< 6). After splitting S_1, by adding two line segments l_1 and l_2 (dashed lines), p_1 falls inside partition $S_1^{'}$ and the enclosing circle becomes O_3. O_3's diameter is $\sqrt{34}(< 6)$, therefore c_1 is *satisfied*.

If we further partition S_2, which contains p_2, c_1 remains *uncertain* because S_2 is bounding neither O_1 nor O_2. Merging all the parti-

tions in S_1 renders c_1 *uncertain* again. However, merging S_2 and S_3 does not have any effect on c_1. The partition splitting process may affect the result of the constraint when the partition bounds intersect either the enclosing circle or the intersecting circle (i.e., S_1 bounds both circles). The partition merging process may affect the result of the constraint when the partition after a merge is intersected by the enclosing or intersecting circle (i.e., S_1 intersects with O_3). We call a constraint *split affected* w.r.t. a certain partition if the constraint is *uncertain*, and has some bodies involved inside the partition, and the partition bounds either the intersecting circle or the enclosing circle. A constraint is *merge affected* w.r.t. a certain partition if the constraint is *satisfied* or *unsatisfied*, has some bodies involved inside the partition that is being merged, and the merged partition (a second level node) is intersected by either the enclosing or the intersecting circle.

In the above example, c_1 is *split affected* w.r.t. S_1 and *merge affected* w.r.t. $S_1^{'}$. Since the *split affected* constraints are the only constraints whose results might change in the split adjustment, when computing the reduction of the *uncertain* constraints after the splitting (Line 3 in Split_Adjust), only the *split affected* constraints are considered because no other constraint changes its result due to the split. Similarly, only *merge affected* constraints have to be considered (Line 3 in Merge_Adjust) for counting the increase in *uncertain* constraints after a merge. This greatly reduces the overhead of recounting in the split and merge processes.

Also, since the purpose of splitting is to reduce the *uncertain* constraints, the candidate nodes for a split are selected such that they contain the largest number of *split affected* constraints (Line 1 in Split_Adjust). This improves the performance of the partition adjustment because only the constraints that may be affected by the split are counted. The purpose of merging is to reduce the partition update rate, therefore the candidate nodes (second level partition S_i) for merge are selected such that these nodes contain the largest estimated partition update rate, PUR_{S_i} (Line 1 in Merge_Adjust.)

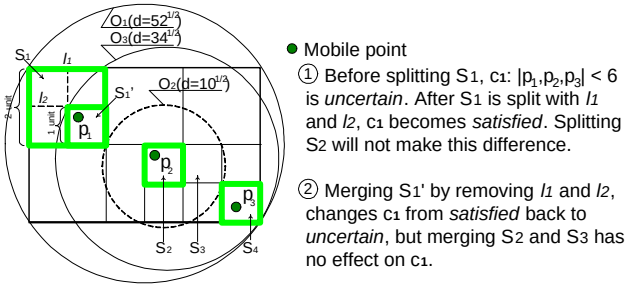

Figure 4: Split/Merge Affected Constraints

2.3 Conflicting and Harmonious Constraints

There can be constraints that are in *conflict* with each other. For instance, if $c_1(|{p_1}^t, {p_2}^t| > d)$ is *satisfied*, $c_2(|{p_1}^t, {p_2}^t| < d)$ must be *unsatisfied*. Similarly, constraints can also be in *harmony* with each other. If a constraint is *satisfied* (*unsatisfied*), this could imply that another constraint is *satisfied* (*unsatisfied*). For instance, if $c_3(|{p_1}^t, {p_2}^t, {p_3}^t| < d)$ is *satisfied*, then it follows that $c_4(|{p_1}^t, {p_2}^t| < d)$ is also *satisfied*. The conflicting and harmonious relation is asymmetric. c_2 is *unsatisfied* might not indicate that c_1 is *satisfied*. Also, even if c_4 is *satisfied*, c_3 might still be *unsatisfied*. For the discussion below, we assume constraint c_1 is associated with object set P_1 and constraint c_2 is associated with object set P_2 etcetera.

Conflicting and harmonious relations for n-body constraints are identified with a set of rules summarized in Table 1 (rules for n-body static constraint can be derived in a similar fashion [25]). The first and second column in the table are the description of two constraints, the third column are the conditions based on which the conflicting and harmonious relation is identified. The last column specifies the transit condition between the two constraints. For instance, the relation among c_3 and c_4 (if c_3 is *satisfied*, c_4 is also *satisfied*) is identified with the rule in Line 1, because c_3 and c_4 have the same alerting distance and the object set for c_3 is a superset of the object set for c_4.

Table 1: Conflicting and Harmonious Rules (Explanation in Text)

c_1	c_2	conditions	transit condition
$P_1 < d_1$	$P_2 < d_2$	$d_1 \leq d_2, P_1 \supseteq P_2$	c_1 is *satisfied* $\rightarrow$
$P_1 > d_1$	$P_2 > d_2$	$d_1 \geq d_2, P_1 \subseteq P_2$	c_2 is *satisfied*
$P_1 < d_1$	$P_2 > d_2$	$d_1 \leq d_2, P_1 \supseteq P_2$	c_1 is *satisfied* $\rightarrow$
$P_1 > d_1$	$P_2 < d_2$	$d_1 \geq d_2, P_1 \subseteq P_2$	c_2 is *unsatisfied*
$P_1 > d_1$	$P_2 < d_2$	$d_1 < d_2, P_1 \supseteq P_2$	c_1 is *unsatisfied*
$P_1 < d_1$	$P_2 > d_2$	$d_1 > d_2, P_1 \subseteq P_2$	$\rightarrow c_2$ is *satisfied*
$P_1 < d_1$	$P_2 < d_2$	$d_1 \geq d_2, P_1 \subseteq P_2$	c_1 is *unsatisfied*
$P_1 > d_1$	$P_2 > d_2$	$d_1 \leq d_2, P_1 \supseteq P_2$	$\rightarrow c_2$ is *unsatisfied*

For the condition column, it must hold that the object set of one constraint is the subset or the superset of the object set of another constraint. If P_1 is not a subset or a superset of P_2 and yet their intersection $P_{1,2}$ (=$P_1 \cap P_2$) contains more than two objects, there is still a potential dependency between these objects. Consequently, the result of one constraint may be partially valuable for the evaluation of other constraints. To fully exploit the correlation of constraints with intersecting object sets, the notion of a *virtual constraint* is developed. A virtual constraint, c_v, involves the intersecting objects $P_{1,2}$ (=$P_1 \cap P_2$). The rules that define the generation of c_v are given in Table 2. ϵ below denotes the smallest positive value allowed by the computer. It is hardware dependent.

Table 2: Virtual Constraint Generation Rules

c_1	c_2	c_v	transit condition
$P_1 > d_1$	$P_2 > d_2$	$P_{1,2} > \texttt{max}(d_1, d_2)$	c_v is *satisfied* $\rightarrow c_1$ and c_2 are *satisfied*
$P_1 < d_1$	$P_2 < d_2$	$P_{1,2} > \texttt{max}(d_1, d_2)$	c_v is *satisfied* $\rightarrow c_1$ and c_2 are *unsatisfied*
$P_1 > d_1$	$P_2 > d_2$	$P_{1,2} < \texttt{max}(d_1, d_2) + \epsilon (\epsilon > 0)$	c_v is *unsatisfied* $\rightarrow c_1$ and c_2 are *satisfied*
$P_1 < d_1$	$P_2 < d_2$	$P_{1,2} < \texttt{max}(d_1, d_2)$	c_v is *unsatisfied* $\rightarrow c_1$ and c_2 are *unsatisfied*
$P_1 > d_1$	$P_2 < d_2$	$P_{1,2} > \texttt{max}(d_1, d_2)$	c_v is *satisfied* $\rightarrow c_1$ is *satisfied*,but c_2 is not
$P_1 > d_1$	$P_2 < d_2$	$P_{1,2} < \texttt{max}(d_1, d_2) + \epsilon (\epsilon > 0)$	c_v is *satisfied* $\rightarrow c_1$ is *satisfied*,but c_2 is not

To amortize the evaluation cost, a secondary graph structure is used to index conflicting and harmonious constraints. This graph structure is constructed and incrementally updated when new constraints are inserted or removed from the system. At the start, to construct the graph structure, each constraint (including virtual constraints) are regarded as nodes in the graph. Each constraint connects to its associated conflicting and harmonious constraints (other nodes) with unidirectional links (representing the asymmetric relation). Each link is appropriately labeled with a transit condition, for example, *satisfied* $\rightarrow$ *unsatisfied* (shorthanded as $s \rightarrow u$) etcetera. The nodes with the same neighbors and same transit condition labels are merged into a compound node. If a constraint is removed, the corresponding node may need to be deleted.

In the evaluation stage, after the result for one of the indexed constraints is computed, a breadth-first transversal of the graph starting from that constraint is performed, immediately giving rise to the results of associated constraints (i.e., represented by linked nodes in the graph.) Virtual constraints are evaluated based on

adaptive space partitioning, just as an ordinary constraint, but they have higher evaluation priority and will always be evaluated before the original constraints based on which they are constructed. Conflicting and harmonious constraint management is orthogonal to partition-based pruning performed by the indexes. The graph index can prune *uncertain* constraints which space partitioning is incapable of pruning.

In Fig. 5, we give an example of conflicting or harmonious constraints represented by our graph index. The five constraints being indexed are shown in node N_1, N_3, N_4 N_5 and N_6. The constraints in node N_3 and N_4 share the same objects p_2 and p_3, therefore a virtual node N_2 representing constraint $|p_2, p_3| > d$ is constructed (rule in Line 2 of Table 2). If the virtual node N_2 is *satisfied*, N_3 and N_4 are *unsatisfied* (unidirectional edge $s \rightarrow u$ from N_2 to both N_3 and N_4). Also, if N_3 or N_4 is *satisfied*, then N_2 must be *unsatisfied* (unidirectional edge $s \rightarrow u$ from both N_3 and N_4 to N_2). Since the two unidirectional edges between N_2 and N_3 (N_4) have the same label ($s \rightarrow u$), they are combined as a single bi-directional edge. Observe that N_3 and N_4 are connected with the edges of the same label to other nodes, therefore they can be combined to form a compound node. All the other transit edges are based on the rules listed in Table 1.

An example of constraint processing is that when the object p_2 updates its location, the virtual constraint $|p_2, p_3| > d$ in N_2 will be evaluated first (given it is *uncertain*) because a virtual node has higher evaluation priority. If it is *unsatisfied*, then the constraint in N_1 is *satisfied* because the transit condition from N_2 to N_1 is $u \rightarrow s$. Since there is no edge coming out from N_1 labeled with $s \rightarrow *$, the transversal terminates. On the other hand, if the constraints in N_2 is *satisfied*, then constraints in N_3, N_4 and N_5 must be *unsatisfied* ($s \rightarrow u$).

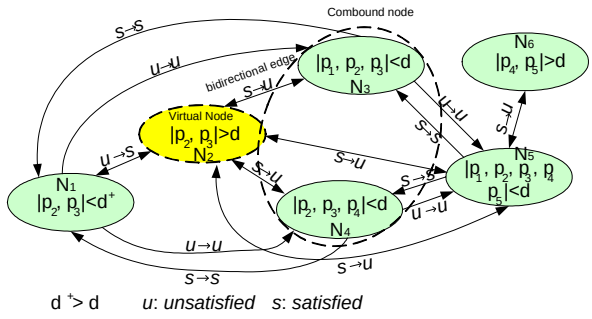

Figure 5: Index for Conflicting and Harmonious Constraints

2.4 Location Update Policy

The location update policy manages the consistency between the last position assumed by the constraint solver and the actual position of the moving object. Each location update exerts certain load onto the system (e.g., network, power, compute resources, and economical. The *location update policy* is the rule that specifies when the update should be scheduled. The policy balances the trade-off between the resources consumed and the accuracy represented in the system and achievable in the results.

In practice, there are many different ways to obtain location data and power is a concern that requires the location update policy to be adjusted to the characteristics of the mobile device. Here, we adopt the notion of a safe region to define a device-dependent location update policy [9]. The objectives are to maintain up-to-date constraint results, and reduce wireless communication and re-evaluation cost.

We assume that the mobile devices are able to track their position and are capable of simple computation for checking whether coordinates are inside a certain partition (rectangular region).[2]

If the current result of the constraint based on the partition information is *satisfied*, we set a circle with the alerting distance as the diameter such that it covers the maximal number of partitions including all the partitions containing involved objects. Then, the partitions covered by this circle are regarded as a safe region and returned to the device, which does not need to update its location as long as it is moving within the safe region, because in this case the constraints are always *satisfied* (see Fig. 6(A)).

If the result is *unsatisfied*, we first identify the objects p_i and p_j (called *sentinel objects*) that are located inside two partitions with a distance greater than the alerting distance. Then, a *bisecting stripe* is set. It is defined as a stripe with width equal to the alerting distance. It bisects the distance between the two partitions (see Fig. 6(B)). As long as each *sentinel object* is moving within the partitions on its side of the stripe (regarded as safe region), the constraints are *unsatisfied*. Therefore, p_i and p_j track their own position and do not need to update their location unless they go beyond the safe region on their sides. At the same time, all the other objects wait for their position probe. Upon notification that either p_i or p_j are outside the safe region, the constraint matcher forces all the involved objects to update their locations, and validates the new *sentinel objects* and the stripe. It is possible that such *sentinel objects* do not exist. In this case, the device updates its location with some default rate. This safe region policy is very cost effective in that it captures most constraint matches with significant reduction of the required location updates.

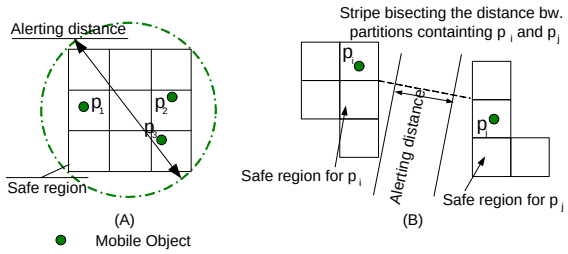

Figure 6: Safe Region Computation

2.5 Extended Constraints

In this subsection we illustrate how the partition-based evaluation algorithm can be applied to other constraints. The illustration is based on a continuous reverse range query (CRR) and a continuous reverse k-nearest neighbor query ($CRKNN$), which are defined as follows:

1. Given a rectangular region R, a Continuous Reverse Range query, $CRR(R, p_i)$ continuously checks whether a moving objects p_i is inside R.

2. Given a static point A, a Continuous Reverse k-Nearest Neighbor query, $CRKNN\ (A, p_i)$, continuously checks whether A is one of the k nearest neighbors of p_i.

In the discussion below we refer to these queries as constraints to emphasize the continuous character of the problem, i.e., given a potentially large set of constraints and changing object location information, we are looking for the matched constraints. As in the previous subsection, the objective of the partition-based evaluation

[2]The device could either obtain its location via GPS or request it from the networking infrastructure. Both are viable models in practice.

is to reduce the number of constraints that have to be evaluated against precise position information.

Without loss of generality, suppose that p_i is located inside partition S_{p_i}. The partition-based evaluation for $CRR(R, p_i)$ works as follows. If S_{p_i} is disjoint with R, the constraint is *unsatisfied*. If S_{p_i} is completely enclosed by R, the constraint is *satisfied*. Otherwise (S_{p_i} intersects R), it is *uncertain*. Fig. 7 shows three range queries, R_1, R_2 and R_3. Since partition S_{53}, which accommodates p_1, is disjoint with R_1, the constraint $CRR(R_1, p_1)$ must be *unsatisfied*. $CRR(R_2, p_1)$ is *satisfied* since S_{53} is completely inside R_2 and $CRR(R_3, p_1)$ is *uncertain* since R_3 partially intersects S_{53}.

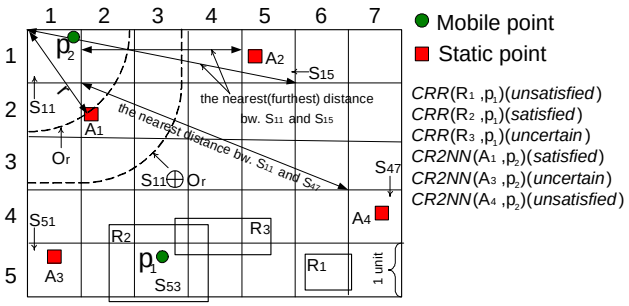

Figure 7: CRR Query and CRKNN Query

The partition-based evaluation for $CRKNN(A, p_i)$ works as follows. Suppose A is inside the partition S_A and p_i is located in S_{p_i}. The system maintains a current list of kNNs and the partitions where they are located(suppose they are $S_{l_j} (1 \leq j \leq k)$.) If the nearest distance between S_{p_i} and S_A is greater than the furthest distance between S_{p_i} and S_{l_j}, or more formally, $inf_{p \in S_{p_i}, q \in S_A} ||p, q||_2 > max_{1 \leq j \leq k} \ sup_{p \in S_{p_i}, q \in S_{l_j}} ||p, q||_2$, then A can not be one of the k nearest neighbors of p_i, and $CRKNN(A, p_i)$ is *unsatisfied*. Let $r = sup_{p \in S_{p_i}} ||p, A||_2$ and O_r is a circle with r as the radius. Then if the *Minkowski* sum of S_{p_i} and $O_r (S_{p_i} \oplus O_r)$ [3] contains at most k neighbors, $CRKNN(A, p_i)$ must be *satisfied*. If a constraint is neither *unsatisfied* nor *satisfied*, it is *uncertain*.

In Fig. 7, suppose the current two nearest neighbors of moving object p_2 are A_1 and A_2. Since $inf_{p \in S_{11}, q \in S_{47}} (p, q) = \sqrt{29} > max(\ sup_{p \in \{S_{22}, S_{15}\}, q \in S_{11}} \ ||p, q||_2) = \sqrt{26}$, A_4 cannot be one of p_2's 2 nearest neighbors, $CR2NN(A_4, p_2)$ is *unsatisfied*. However, constraint $CR2NN(A_1, p_2)$ must be *satisfied*, because A_1 is the only neighbor inside the region $S_{11} \oplus O_r$ where $r = sup_{p \in S_{11}} ||p, A_1||_2$. And it can be verified that the constraint $CR2NN(A_3, p_2)$ is *uncertain*. In fact, if p_2 is moving to the bottom-left of S_{11}, A_3 will replace A_2 to be the second nearest neighbor of p_2. The result of constraint $CR2NN(A_3, p_2)$ is truly undetermined unless the precise position of p_2 is given.

Bear in mind that for location change within the partition, explicit computation is only required for the constraints previously classified as *uncertain* and the constraint classification only happens in response to a partition update.

3. ANALYTICAL MODEL

In this section, we develop an analytical model for the cost of evaluating n-body constraints. The analysis for n-body static constraints is similar. In our model, a total of c constraints is associated with a total of o moving objects. Suppose that each constraint is associated on average with $\overline{o}$ objects and among the c constraints, u are *uncertain*. The model estimates the matching cost for in-memory and I/O-incurring environments and formalizes algorithmic properties. Due to space limitations, the proofs of all the theorems and lemmas below are omitted. They can be found in the technical report [25].

[3] $M_1 \oplus M_2 = \{p + q | p \in M_1, q \in M_2\}$, where $p + q$ is the vector sum of the vectors p and q.

3.1 Cost for In-memory Processing

All computations are done in main memory and no disk I/O is incurred. On average each object is associated with $c\overline{o}/o$ constraints and $u\overline{o}/o$ of them are *uncertain*. Since the frequency of the partition adjustment processes is much lower than the location update frequency, the cost for maintaining the index data structure is ignored. The constraint evaluation cost is comprised of two factors: the cost for evaluating a constraint based on precise location position information (denoted as $Cost_{lu}$) and the partition-based evaluation cost when an object updates its partition (denoted as $Cost_{pu}$). Suppose that for a time duration, t, a total of l location updates are received, p of which incur partition updates. If each evaluation based on new location data takes approximately α processing time, then the cost for a location update during one unit of time equals $\frac{(u\overline{o}/o)\alpha l}{t}$. Note that $\frac{l}{ot}$ is the average update frequency (denoted as $\overline{f}$). With this, the above location update cost can be simplified to:

$$Cost_{lu} = u\overline{o}\alpha\overline{f} \quad (1)$$

When the object is moving into a new partition, all the constraints it is associated with are subjected to partition-based evaluation. If partition-based evaluation incurs β processing time, the cost of a partition update per unit of time is $\frac{(c\overline{o}/o)\beta p}{t}$. p/l is the probability of an object updating its partition (denoted as P_{pu}) and the partition update cost can be rewritten as:

$$Cost_{pu} = c\overline{o}\beta\overline{f}P_{pu} \quad (2)$$

The value of α and β depend on the hardware used and can be estimated experimentally. From Eq. 1 and Eq. 2, it follows that the average cost per unit of time for the adaptive algorithm is given by:

$$Cost_{adapt} = Cost_{lu} + Cost_{pu} = u\overline{o}\alpha\overline{f} + c\overline{o}\beta\overline{f}P_{pu} \quad (3)$$

The per unit of time cost for the naïve approach is simply the cost for evaluating all constraints:

$$Cost_{naïve} = c\overline{o}\alpha\overline{f} \quad (4)$$

In Eq. 3, if P_{pu} is small, the second term can be ignored, and the cost can be approximated to:

$$Cost_{adapt} \approx u\overline{o}\alpha\overline{f} = \frac{u}{c}Cost_{naïve} \quad (5)$$

Discussion: Our algorithm outperforms the naïve approach when $Cost_{adapt} < Cost_{naïve}$. However, with static space partitioning, this can not be guaranteed. For instance, if the majority of the constraints are *uncertain* (larger u) or P_{pu} is too high, then the performance of the algorithm degenerates. Given a static space partitioning, P_{pu} depends only on the movement pattern (e.g., the velocity and the position) of the objects while the number of *uncertain* constraints depends on the correlation of the position of objects. There is no direct relationship between P_{pu} and the number of *uncertain* constraints. Also, deriving the optimal partition scheme is difficult, because there is no direct dependency between P_{pu} and the partition scheme. For example, even if the space is partitioned sparsely, the object could still generate a large number of partition updates by moving back-and-forth across a splitting line. On the other hand, an object could move fast without generating any partition updates in a densely partitioned region by remaining inside a single partition. We address this challenge in the next subsection.

3.2 Determining the Optimal Partition Size

In this section we develop a model to estimate the optimal partition size. The idea is to sample a number of partition schemes and

select the best one (i.e., the one that leads to a smaller $Cost_{adapt}$.) This is what our algorithm does by tentatively splitting and merging partitions to determine a partition change that ultimately reduces the overall cost. The change aims to reduce both P_{pu} and the number of *uncertain* constraints, and aims to balance the trade-off between them. Our analytical model is based on the following assumptions:

1. All objects follow the random movement pattern. As a result, the distribution of the objects on the plane is uniform.
2. The partitions are equal-sized squares with the length of each side equal to a, which is much smaller than the extent [4] of the whole space s ($a \ll s$).
3. All the constraints have a unique alerting distances, denoted as d ($d \ll s$).

First, we establish that the size of the diameter of the smallest circle enclosing a set of objects is uniformly distributed. Therefore, the probability of a constraint being *uncertain* (*satisfied* or *unsatisfied*) is an expression of the partition size, space size and the alerting distance. This is captured in Lemma 1.

LEMMA 1. *Under random movement of objects in Euclidean 2D space, the distribution of the size of the diameter of the smallest circle enclosing a set of objects is uniform. Therefore, a constraint is unsatisfied with probability* $1 - \frac{a\lceil d/a \rceil}{s}$*, is satisfied with probability* $\frac{a\lfloor d/a \rfloor}{s}$ *and is uncertain with probability* $\frac{a}{s}$.

Based on Lemma 1 and the aforementioned assumptions, we can estimate the optimal size of the partition that minimizes the evaluation cost. This is stated in the following theorem.

THEOREM 1. *Under random movement of objects in Euclidean 2D space, there exists a unique local (therefore also global) optimal partition size, which is* $\frac{k\overline{v}\beta s}{\alpha}$ *for some constant* k*. (recall that* α *and* β *are the average cost for evaluation based on precise position and partition information, respectively.)*

Discussion: If the movement pattern is not random, we can not compute a globally optimal partition size. However, if we interpret a local distribution as random, then there still exists a local optimum, which gives us a locally optimal partitioning. We approximate this local optimum by rescheduling the partitioning in the area that is not yet optimized based on the merging and splitting algorithm, introduced in the previous section.

We say a partition scheme, ps, is the ancestor of a partition scheme, ps', (ps' is the descendant of ps), if ps' can be reached from ps by adding splitting lines. This descendant relationship is denoted as $ps' \succ ps$. In Section 2.2, to reduce the computation, we mentioned that the *uncertain* constraints after the splitting (merging) process must be a subset (superset) of the *uncertain* constraints before the splitting (merging). This property is formally captured in Lemma 2. Proof details can be found in the technical report [25].

LEMMA 2. *If* $ps' \succ ps$*, then an unsatisfied constraint in* ps *is also an unsatisfied constraint in* ps'*; a satisfied constraint in* ps *is also a satisfied constraint in* ps'*; and an uncertain constraint in* ps' *is also an uncertain constraint in* ps.

The intuition behind Lemma 2 is that with a more precise approximation (with ps'), more constraints can be resolved as *satisfied* and *unsatisfied* by the partition-based evaluation. The splitting process aims at a reduction of the number of *uncertain* constraints at the cost of a slight increase in the partition update rate. On the other hand, the merging process aims at a reduction of the partition update rate at the cost of a slight increase in the number of *uncertain* constraints. By continuous adjustment, the algorithm adapts to a non-uniform environment and evolves with the change of the movement patterns and reaches a locally optimal partitioning.

[4]The extent of the space is defined as the maximum possible distance between two points inside the space.

3.3 Cost under Secondary Storage Access

For applications that need to manage a large number of location constraints with a large number of moving objects, secondary storage access cost of the algorithm may become a performance degrading factor. For instance, objects may be associated with large profiles, as is common in telecommunication applications, or the application may be collocated with other applications, thus not have exclusive access to main memory. The I/O cost for accessing data in secondary storage is typically orders of magnitude higher than accessing data residing in main memory and it becomes the major overhead. Therefore reducing the number of secondary storage accesses becomes a primary concern in this environment.

For every secondary storage access, data is fetched in pages rather than in records and each access to secondary storage transfers a constant amount of data into memory. The I/O cost is measured as the number of accesses to the data on the secondary storage device. For optimizing secondary storage accesses, our algorithm aims at holding as many *uncertain* constraints as possible in main memory because they are the most frequently accessed data. Constraints that do not fit into main memory reside on disk and are brought in on demand. The following theorem quantifies the cost (the number of I/O accesses per time unit) of our algorithm relative to the naïve approach. A proof can be found in [25].

THEOREM 2. *If all the uncertain constraints can be held in main memory, the cost of the matching algorithm is the partition update probability (*P_{pu}*) times the cost of the naïve approach (*$Cost^{I/O}_{adapt} = P_{pu} * Cost^{I/O}_{naïve}$*).*

This result is somewhat surprising, it means that even with limited memory, reducing the partition update rate by some proportion causes the evaluation cost to be reduced with the same proportion, given that all *uncertain* constraints are held in memory. Under the random movement pattern, the number of *uncertain* constraints is only a small portion of the total number of constraints ($\frac{a}{s}$ as stated in Lemma 1). It can therefore often fit into main memory, even for large constraint loads going beyond main memory capacities. Also an efficient algorithm should stop further reducing the number of *uncertain* constraints by splitting as long as they all fit into memory. Merging in that case may actually dampen the partition update rate and help to lower the I/O access cost further.

4. SYSTEM IMPLEMENTATION

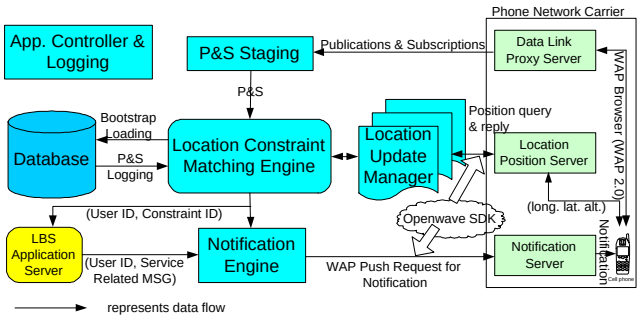

Figure 8: End-to-end System Architecture (fully implemented)

We have developed a complete end-to-end location-based service prototype incorporating the location constraint matcher de-

scribed in this paper. More information about an earlier version of the prototype is summarized in [23] and is known as the Location-based Toronto Publish/Subscribe System (L-ToPSS). The system was deployed as a proof of concept on a mobile cellular network. The overall system architecture is shown in Fig. 8.

In our implementation, the system uses the cellular network to obtain location position information of subscribers. The network exports location tracking capabilities via a Web service. Location position data is retrieved through a location position server over the Internet. Both the server and the subscriber can initiate position requests. When the location update streams into the constraint matching engine, it is evaluated against the location constraints stored in the system and the matches are communicated back to the subscribers via the notification server on the carrier's side. Different solutions for obtaining location position data are available in practice. In our case, the operator combined GPS, network triangulation, and cell site location technologies to position subscribers, aiming to provide the most accurate position for a required precision, specified as part of the location request input. A brief evaluation of the accuracy of the system under different experimental conditions is summarized in [22].

5. EXPERIMENTS

In this section, we present the experimental results that demonstrate the performance of the algorithms. All experiments were conducted on a Pentium 4 with 2GHz and 2G RAM) running under Linux. In the experiments we simulate mobile objects moving in a test field of size $40km \times 40km$. We model two movement patterns, random movement and clustered movement. The random movement pattern maintains a uniform distribution of the objects in the test field. The clustered pattern maintains a number of clusters of objects in the test field, where the position of the object in the cluster follows a normal distribution with mean, μ, as a randomly selected point in the test field. We vary the standard deviation, σ, of the distribution to model different degrees of skewness. In the extreme case, where the standard deviation is sufficiently large, the clustered movement pattern is reduced to the random movement pattern. To model the movement of clusters, μ is determined as a function of time, t.

Prior to the experiment, constraints are generated; each constraint is associated with n bodies among all the moving objects in the field. The alerting distances are uniformly distributed with a certain mean (e.g., 500m). By changing the mean, the *matching load* [5] can be adjusted. The velocity of the moving objects follows a normal distribution with a given mean (5 m/s or as specified).

The partition adjustment is performed once every minute to optimize the partition scheme as needed. For sake of fairness in the comparison, we maintain the same adjustment speed for both indexes by splitting and merging the same number of candidate partitions.

5.1 Effect of Adaptation

This experiment is conducted under random movement pattern where the objects are uniformly distributed. It shows how AKDT and AMLG adapt to the random movement patterns. The results are compared against a non-adaptive solution ($w = 0$). With 100,000 constraints and 10,000 uniformly distributed objects, we measure the matching time and the average partition size over time. Fig. 9(A, B) shows the results during the first 12 minutes of the experiment when the initial partition size is $25km^2$. The static index ($w = 0$) does not evolve at all over time to better accommodate the movement pattern, the matching time for both AKDT and AMLG is around 250ms and the partition size is $25km^2$. When the window size (which controls the speed of adjustment) is adjusted to 25, AKDT and AMLG actively evolve to lower the evaluation load through splitting. Within seven minutes, the matching time of both indexes is reduced by 60% and stabilizes below 100ms, and the partition size is adjusted to around $5km^2$. This performance gain comes about through the pruning of large numbers of *uncertain* constraints (by about 60%) at the cost of a moderate increase in partition updates (to about 5%) (graphs omitted due to space limitation). This shows that the adaptive algorithms find the optimal partition size through self-adjustment.

On the other hand, if the initial partition size is set to $1km^2$ (Fig. 9(C,D)), partition adjustment ($w = 25$) evolves to merge the partitions to an average size of about $5km^2$. The average matching time is reduced from 350ms to below 100ms (by about 70%). This performance gain is due to the reduction in the partition update rate (by about 90%) of the merge process. The matching time resulting from the naïve approach, where all the constraints are evaluated without partition-based pruning, is an order of magnitude higher (not shown in the graphs for clarity.)

In Fig. 10(A), we vary the mean of the velocity ($v = 5, 10, 15$) and run the data set against adaptive indexing with different window sizes. (For clarity of presentation, we only show results for the AKDT; results for AMLG are similar.) The figure shows that the adjustment cost (the overhead of splitting and merging) increases linearly as the window size increases for different velocities. However, the adjustment cost is negligible compared to the time for constraint evaluation. To obtain the same evaluation time in the three scenarios shown, a larger window size is required for higher velocity.

Given the distribution of the velocity, the optimal partition size can be found with the steepest descent method (SDM [7]) by sampling different partition sizes approaching the optimum, which based on Theorem 1 is unique. In Fig. 10(B), we validate the relationship between the average velocity of the objects and the optimal partition size and show that the optimal partition size obtained with SDM is nearly proportional to the average velocity of the objects (as confirmed by Theorem 1). Moreover, the figure shows that the adaptive processing eventually results in a partition scheme that is close to the optimum.

In Fig. 10(C), we evaluate the adaptation of the algorithm with the evolution of the clustered movement pattern. 50 clusters are generated with constant standard deviation (400), the means of these clusters are moving with random speed (uniform in the range [20,50]) and direction. As evident from the moving cluster workload, the static index cannot keep up and performance deteriorates. The matching time is not only high but also very unstable with many spikes resulting from large partition update overhead when the clusters move across splitting lines. AKDT and AMLG handle this situation well. The matching time is relatively low and much more stable. When we adjust the skewness of the cluster (by varying the standard deviation σ), we obtain similar results, except that higher skewness (smaller σ) exacerbates the instability for the static indexing. Fig. 10(D) shows the superiority of the adaptive indexing for the CRR and $CRKNN$ constraints. Compared with the basic processing without constraint pruning, the matching time is greatly reduced (to 14% for $CRKNN$ and to 9% for CRR) for the adaptive partition-based approach.

Fig. 11(A) demonstrates the cost of indexing for different location update loads (i.e., number of location updates incurred.) With static indexing ($w = 0$), for AKDT with backtracking op-

[5]The matching load is the ratio between the average number of *satisfied* constraints and the total number of constraints.

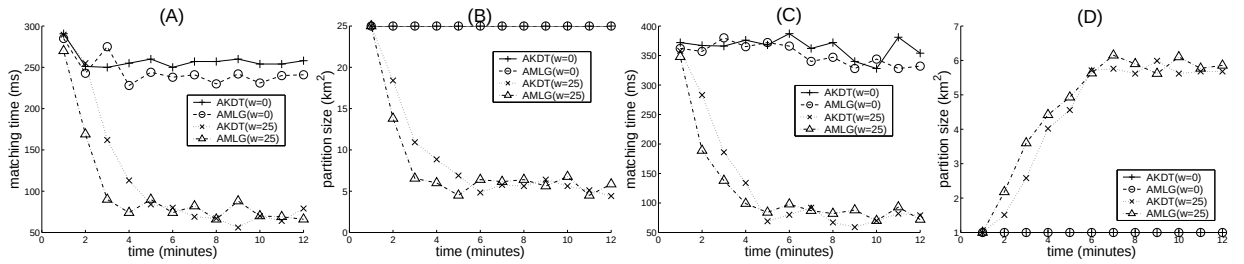

Figure 9: Adaptation to the Uniform Movement Pattern

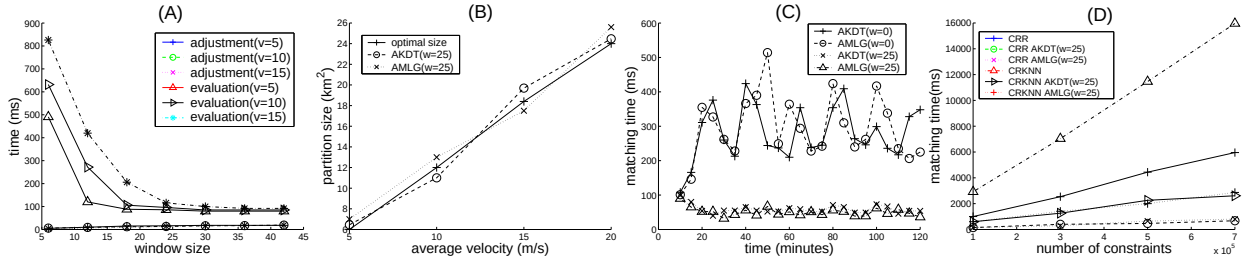

Figure 10: Adjustment Cost (A), Optimal Partition Size (B), Adaptation to Clustered Pattern (C), and CRR/CRKNN queries (D)

timization, [6] the overhead is reduced by 50% as compared to non-backtracking-based indexing. For adaptive indexing ($w = 25$), the cost of AKDT (with or without backtracking) is very close to that of AMLG, because adaptive space partitioning avoids high partition update rates and the maintenance difference between AKDT and AMLG is small. Both cost are about 10% of the cost of static indexing.

We also observe in Fig. 11(B) that the matching time decreases linearly as the percentage of the conflicting and harmonious constraints increases. These constraints are pruned directly through the secondary graph-structured index and do not need to be re-evaluated.

Fig. 11(C) shows the adaptation of the algorithms to different constraint sets. Three constraint sets are deliberately generated such that for static indexing ($w = 0$), the sets contain different number of *uncertain* constraints (set 1 $<$ set 2 $<$ set 3), therefore, we observe an increase in matching time from constraint set 1 to set 3. With adaptive indexing ($w = 25$), the matching time stabilizes around 100ms for all sets, because the partition scheme evolves to reduce the *uncertain* constraints to the same level.

Now, with all the constraint matches detected, we compare the number of location updates issued by the safe region policy with the frequency-based policy. The location update frequency of the frequency-based policy is adjusted (to around 1/s) such that all the matches are detected (with lower frequency, some matches are missed). Fig. 11(D) shows that, for all three constraint sets, the safe region policy is the most cost effective in that it detects the same number of matches with only 10% of the location update cost of the frequency-based policy. This is a significant saving that can directly translate to dollar value.

[6]The search backtracks level-by-level up the k-d-tree from the original leave node where the object is located until the node is found that contains the object. Insertion proceeds down from this node, rather than from the root of the tree. This approach avoids the top-bottom scan of the tree for a location update.

5.2 Effect of Secondary Storage Access

In this subsection we measure the number of disk accesses required for processing 10,000 objects and 100,000 2-body constraints, with strict limits imposed on the available memory. We assume that the indexing structure, the position information of the objects and a certain number of location constraints are held in memory. Other location constraints have to be paged in and out of memory on demand (we assume the LRU page replacement policy and a page size of $4k$). The algorithm tries to store as many *uncertain* constraints in memory as possible, so that disk access is minimized.

Fig. 12(A) plots the number of secondary storage accesses against available memory. For this experiment, we test the AKDT and AMLG indexes against objects with various movement patterns. Fig. 12(A) shows that the number of disk accesses for both indexes are much lower than for the naïve algorithm. This is because there are fewer constraints that need to be accessed due to the pruning capabilities of the indexes. The partition-based algorithms exhibit a point in memory use after which the number of disk accesses is nearly proportional to the disk accesses for the naïve approach (around 80 pages for $w = 25$, 120 pages for $w = 12$ and 160 pages for $w = 0$). This is the point where the available memory suffices to load all *uncertain* constraints. The adaptive algorithms with window size 25 reduce the *uncertain* constraints (by 60%), therefore less memory is required to store them. If the page number exceeds this amount, the number of disk accesses equals $P_{pu}Cost^{I/O}_{naïve}$, as predicted by Theorem 2. The graph for the adaptive algorithms ($w > 0$) is much flatter than the non-adaptive one ($w = 0$) after this point. Based on Theorem 2, we can deduce that the adaptive algorithms have partition update rates that are much lower (about 60% lower for $w = 25$ and 30% lower for $w = 12$) than the non-adaptive algorithms. Therefore, the adaptive algorithms outperform the non-adaptive ones, also in I/O incurring environments.

Furthermore, in Fig. 12(B), we suppress the splitting process as soon as the memory can hold all the *uncertain* constraints. This means, if memory is available, the algorithm deliberately increases the *uncertain* constraints in order to reduce the partition update rate

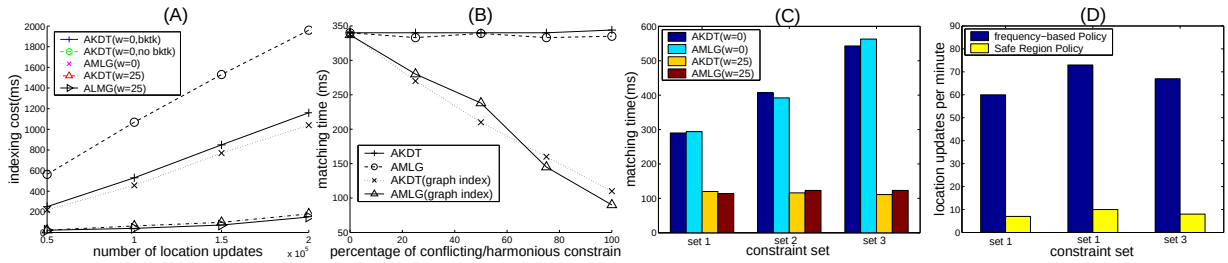

Figure 11: Indexing Cost (A), Graph Pruning (B), Adaptation to Constraint Set (C), and Location Update Policy (D)

(with merge). We observe that suppressing the splitting process reduces the number of I/O accesses even further. The number of I/O access is close to zero after memory capacity exceeds 120 pages ($w = 25$) or 200 pages ($w = 12$). The AMLG case is omitted for the sake of clarity in presentation. It's results are similar.

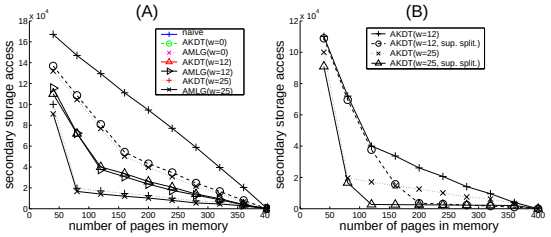

Figure 12: Secondary Storage Access Cost

6. RELATED WORK

In this section we put our work in perspective to constraint databases [12], nearest neighbor problems [8, 15, 26], kinetic data structures [4], and various applications [18, 24]. Most of the related approaches solve a problem very different from the problem addressed in this work.

Constraint databases (CDB) [12] aim at representing large or infinite sets in compact ways. A constraint can be a linear or polynomial equation. CDB are applied to the modeling and integration of spatial and temporal data [6]. However, current work on CDB does not address the efficient indexing of constraints or data to support moving objects that involve frequent updates.

Determining the geometric relationship among, either a set of mobile, or static entities has received wide attention in the literature (for example, [8, 15, 26].) Corral *et al.* [8] study the problem of determining the k closest pairs between two spatial sets of static points. Their approach is similar to a join query and discovers the k smallest distances between two sets of points. Different from location constraint matching, Corral *et al.* do not consider a *close to* relation involving more than two points, the points evaluated are partitioned into two static sets, and distances between pairs of points from the same set are not considered in the evaluation.

The nearest neighbor problem determines the nearest object(s) to a given point among all the objects in the space [15, 26]. This is a different query type from the location constraint matching problem, we are looking at, which determines whether a specific set of objects are in a given spatial constellation to each other at a given point in time.

Continuous range queries [21] are queries that answer which moving objects are currently located inside the boundaries of the query. The authors' implementation takes advantage of incremental changes in object locations to continuously process the range query with the covering tile-based query index (CTQI). The approach differs from our continuous reverse range query (CRR) in that the CTQI retrieves the objects inside the query range, whereas our CRR continuously monitors whether an object is located inside a specific range or not. CTQI is designed to support a small number of range queries over a comparatively large number of moving objects. The complexity of maintaining tiles increases significantly as the number of range queries increases. Our approach is independent of the number of objects and queries (constraints) and is designed to support large numbers of queries and moving objects. Also, the CTQI algorithm is designed for a main memory environment and is not also optimized for I/O incurring environments.

Kinetic data structures (KDS) [4] are used to keep track of dynamic properties of continuously changing data. KDS are based on maintaining and enforcing geometric relations (referred to as certificates) over incremental changes of moving objects. KDS do not employ space or query pruning techniques. Many KDS approaches assume a fixed rate of change (i.e., constant movement velocity), which is difficult to enforce for the application scenarios we target, as future behavior is unpredictable in many scenarios. KDS are often used in scenarios where data change is highly predictable, such as animations.

The buddy tracking application [18] is the only work known to us that is looking at a problem statement similar to the location constraint matching problem. Their solution exclusively looks at a 2-body problem. The approach is based on a distributed algorithm. It assumes that the mobile objects communicate with each other directly to solve a 2-body constraint matching problem by exchanging messages. The optimization objective is to reduce communication cost (i.e., reduce the messages exchanged between mobile entities.) A non-distributed quadtree-based algorithm is also sketched, without an experimental evaluation. The algorithm only works for 2-body constraints and does not support adaptive space partitioning. We therefore expect it to behave similarly to the static space partitioning scheme, used as the baseline in our experiments, which performs poorly under a skewed or clustered movement pattern. Moreover, the solution applied in the buddy tracking application is restricted to one global alerting distance for all registered constraints, which is not feasible for many of the applications we advocate.

In prior work [24] we have built a location-based service and supporting constraint matching infrastructure for processing proximity relations based on a simpler form of the more general location constraints defined in this paper. The system is based on a non-adaptive, static indexing solution that does not adapt to changing movement patterns or constraint load variations and requires knowledge of object clusters in advance to setup the index structures. The approach developed in this paper address these limitations and generalizes the constraint language, also accommodat-

ing conflicting and harmonious constraints in the index. In other prior work [22], we evaluated the precision of available location positioning technologies, and offered a solution for constraint evaluation under position uncertainty. The work focused on developing lower and upper bound for the assessment of the probability of match for a single location constraint. The results of that work complement the indexing approach in this paper, where uncertainty of position information is not considered as a variable in the algorithm.

7. CONCLUSIONS

Location constraint processing is essential for location-based applications that aim at tracking, correlating, and filtering information about moving entities. Application scenarios include friend and family tracking in cellular networks, multi–player online gaming support, security-sensitive area protection in cities and indoor environments, and collision avoidance for air-traffic control. We enable location-aware constraint processing for these kind of applications by defining two classes of location constraints, the n-body constraints and the n-body static constraints, which capture proximity relations among sets of moving objects and sets of moving objects and a static point of demarcation in the environment, respectively. Our constraint language can express *close-by* relations, *no-closer-than* relations, continuous reverse range queries, and continuous reverse kNN queries. Moreover, our constraint language and algorithms support the resolution of conflicting and harmonious constraints registered with the constraint matcher. We develop an adaptive space partitioning approach implemented by the AKDT and the AMLG indexes. Using partition update information derived from the moving objects, only the constraints that are likely *satisfied* (yet *uncertain*) are chosen for further consideration, others are pruned from search by the index.

We experimentally validate our theoretical finding that with a random movement pattern, our adaptive approach determines the optimal space partitioning, with a partition size proportional to the average velocity of the moving objects. The experiments also show that the AKDT and AMLG index adapt themselves to movement patterns and constraint load variations. The resulting partition scheme also approximates the partitioning one can determine manually for a non-adaptive index with a static workload. With an additional graph-structured index for managing conflicting and harmonious constraints, the constraint matching time decreases linearly with the percentage of conflicting and harmonious constraints that can be identified.

We further show experimentally that our approach is well suited for large constraint loads that go beyond capabilities of main memory processing. If main-memory is sufficiently large to hold all *uncertain* constraints, the number of disk accesses is proportional to the partition update probability. Experimental results also show that compared to a non-adaptive index, adaptive indexing reduces the number of *uncertain* constraints and the partition update rate by up to 60%.

In future work, we intend to study the constraint matching problem on road networks where the objects are moving on the edges of a road network graph. In this context the constraint evaluation must use a network distance metric (network hops or link weight) rather than the Euclidian distance.

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/rfid/*.

[3] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *Proc. ACM PODS*, 2000.

[4] Julien Basch. Kinetic data structures. *Ph.D. thesis, Stanford University, Computer Science Dept.*, 1999.

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

[6] Elisa Bertino, Barbara Catania, and Boris Chidlovskii. Indexing Constraint Databases by Using a Dual Representation. In *Proc. ICDE*, 1999.

[7] Richard L. Burden and J. Douglas Faires. Numerical analysis. Brooks/Cole Publishing Company, 2000.

[8] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proc. ACM SIGMOD*, pages 189–200, 2000.

[9] Haibo Hu, Jianliang Xu, and Dik Lun Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *Proc. ACM SIGMOD*, 2005.

[10] D. R. Karger. Finding Nearest Neighbors in Growth-restricted Metrics. In *ACM Symposium on Theory of Computing (STOC)*, 2002.

[11] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. ACM PODS*, 1999.

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

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

[14] S. Prabhakar, Y. Xia, D. Kalashnikov, W. A., and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 2002.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD*, 1995.

[16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. ACM SIGMOD*, 2000.

[17] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R-Tree: A Dynamic Index for Multi-Dimensional Objects. In *The VLDB Journal*, 1987.

[18] A. Amir. A. Efrat. J. Myllymaki. L. Palaniappan. K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. In *INFOCOM*, 2004.

[19] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. In *ACM Transactions on Information Systems (TOIS)*, 1992.

[20] E. Welzl. Smallest Enclosing Disks (Balls and Ellipsoids). In *New Results and New Trends in Computer Science*. Springer, 1991.

[21] Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu. Efficient Processing of Continual Range Queries for Location-Aware Mobile Services. In *Information Systems Frontiers*, 2005.

[22] Z. Xu and H. A. Jacobsen. Evaluating proximity relations under uncertainty. In *Proc. ICDE, 2007*.

[23] Z. Xu and H. A. Jacobsen. Efficient constraint processing for highly personalized location based services. In *Proc. VLDB04*, 2004.

[24] Z. Xu and H. A. Jacobsen. Efficient constraint processing for location-aware computing. In *6th International Conference on Mobile Data Management*, 2005.

[25] Z. Xu and H. A. Jacobsen. Proximity Relation Processing With Evolving Environment. Technical Report, University of Toronto, www.cs.toronto.edu/∼zhengdao/report/CSRG-552.pdf, 2007.

[26] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-Nearest Neighbor Queries over Moving Objects. In *Proc. ICDE*, 2005.

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

[28] Y. Zhao. Standardization of mobile phone positioning for 3G systems. *IEEE Communication Magazine*, 2002.