Workload-Aware Indexing of Continuously Moving Objects

Kostas Tzoumas Man Lung Yiu Christian S. Jensen
Department of Computer Science, Aalborg University, Denmark
{kostas,mly,csj}@cs.aau.dk

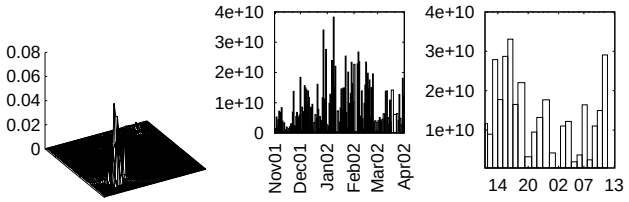
ABSTRACT

The increased deployment of sensors and data communication networks yields data management workloads with update loads that are intense, skewed, and highly bursty. Query loads resulting from location-based services are expected to exhibit similar characteristics. In such environments, index structures can easily become performance bottlenecks. We address the need for indexing that is adaptive to the workload characteristics, called workload-aware, in order to cover the space in between maintaining an accurate index, and having no index at all. Our proposal, QU-Trade, extends R-tree type indexing and achieves workload-awareness by controlling the underlying index's filtering quality. QU-Trade safely drops index updates, increasing the overlap in the index when the workload is update-intensive, and it restores the filtering capabilities of the index when the workload becomes query-intensive. This is done in a non-uniform way in space so that the quality of the index remains high in frequently queried regions, while it deteriorates in frequently updated regions. The adaptation occurs online, without the need for a learning phase. We apply QU-Trade to the R-tree and the TPR-tree, and we offer analytical and empirical studies. In the presence of substantial workload skew, QU-Trade can achieve index update costs close to zero and can also achieve virtually the same query cost as the underlying index.

1. INTRODUCTION

Advances in geographical positioning and wireless communication technologies combine to enable the tracking of the continuously changing positions of mobile objects. These capabilities enable the delivery of novel, location-based services to users with mobile devices, such as mobile phones and online navigation systems. The needs for the scalable delivery of such services to large populations of users yield new data management challenges that call for new technologies that enable the efficient storage, update, and querying of the locations of large populations of moving objects.

The positional *update* workloads that result from the tracking of moving objects are highly skewed in three dimensions. First, there exists a spatial skew. Figure 1(a) shows the probability of receiving an update at the central server as a function of latitude and longitude for the AKTA data set, from a road pricing experiment in Copenhagen [21]. It is obvious that the update workload is far from uniformly distributed across space. Second, there is temporal skew. Figure 1(b) shows the update rate, r_U, per day (in updates per hour), as received by the central server, for a total of five months for the AKTA data set, and Figure 1(c) shows the evolution of the update rate (in updates per minute) within a single day[1]. The update rate can vary by two orders of magnitude and can be highly bursty, both in short and long term observations. Finally, there is a spatio-temporal skew. A recent empirical study shows that human mobility patterns follow a truncated power law, thus offering evidence of substantial skew as a fundamental characteristic of human motion [13]. The *query* workloads produced by location-based services are expected to exhibit skew in the same dimensions.



(a) Spatial histogram (b) r_U per day (c) r_U within a day

Figure 1: Skew of the AKTA data

The maintenance of index structures in such dynamic environments may become a system bottleneck. Any self-tuning mechanism [3] would not recommend the use of an index on continuously changing locations, due to the high update load at peak update rate points. As a result, several works have proposed index structures with a lower update-footprint, often at the expense of higher query cost. These structures may be suitable in time periods when the incoming workload (a mixed stream of updates and queries) is update-intensive (e.g., at the peak points of Figure 1(b)). In contrast, index structures with good query performance would be more suitable during the remaining time periods. However, for the applications we are considering, it is impossible to make such assumptions for the incoming workload, as it is fundamentally skewed and changes in a bursty manner. Rather, an adaptation mechanism has to be provided that automatically changes the behavior of the index according to the workload. We present QU-Trade ("Query-Update Trade"), a layer that can be built on top of any R-tree or TPR-tree based index to achieve this adaptation.

Figure 2 illustrates the notion of *workload-awareness*. The axes capture index update and query performance. An index has specific update and query operators with specific costs. Thus, an index is represented as a point in the figure, and an index represents a fixed

[1]The numbers reported are based on data from 20 randomly chosen cars.

VLDB '09, August 24-28, 2009, Lyon, France

tradeoff between update and query performance. While the figure is intended to be conceptual, the points that represent indexes are consistent with our experimental results. Typically, index structures in the native space such as the R-tree and the TPR-tree excel at query performance, but suffer from poor update performance. We thus depict them in the upper-left corner in. These indexes are ideal for query-intensive workloads. In contrast, grid-based indexes and indexes based on the B-tree typically exhibit much better update performance, at the expense of the query performance. These indexes offer better support for update-intensive workloads. The extreme case of not having an index yields the best index update performance, but also the worst query performance. Because any particular index represents a point in the figure, a specific query-update mix exists for which it is optimal.

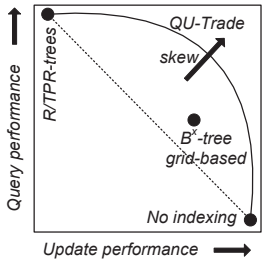


Figure 2: The query-update tradeoff space

QU-Trade aims to overcome this limitation. In particular, QU-Trade defines a curve (the dotted line in the figure) in the query-update tradeoff space. The query and update costs are not fixed, but adapt to the incoming workload. When the workload is update-intensive, the update performance is high, at the expense of query performance. The inverse is true for query-intensive workloads.

Further, QU-Trade approaches optimal performance when the workload skew increases (the solid line in the figure). This is achieved via non-uniform query and update costs with respect to the indexed population. If an object is updated often, the update cost associated with that particular object is low. Conversely, for objects that are queried frequently, the query cost associated with them is low.

Workload-awareness is achieved by controlling the *filtering capability* of the underlying index. QU-Trade is implemented as a layer above the *query* and *update* interface of a traditional R-tree or TPR-tree. It adapts to the workload by changing the amount of overlap in the index structure. Objects are not represented as multi-dimensional points, but as rectangles with extent (termed object windows). These windows are time-parameterized in the case of the TPR-tree. An object's entry in the index is updated only when it moves outside its window. Further, an index update causes the object's window to grow, in order to reduce subsequent updates from the same object. This causes the overlap in the index to increase, yielding higher query cost. When a query comes, the index is used to retrieve a set of candidate objects. This set, depending on the index's quality, may contain a number of false positives. The windows of these false positives are shrunk in order to reduce the overlap in the index, with the goal of reducing the cost of subsequent queries. The competition of window growing and shrinking leads to good average performance and fast online adaptation when the workload changes. Further, the use of local decisions leads to good index quality in frequently queried regions. Quality deteriorates only in regions of space that are not queried often. This is how skew in a workload is exploited. In summary, the salient features of QU-Trade are the following.

1. Workload-awareness. QU-Trade is the first proposal to achieve workload-awareness in the setting of spatial and spatio-temporal indexing.

2. Exploitation of skew. QU-Trade thrives on skewed workloads. Frequently updated objects are associated with a low update cost, and frequently queried regions are associated with a low query cost.

3. Simplicity. QU-Trade lies on top of an R-tree or TPR-tree, so no new index structure has to be implemented. This greatly reduces implementation costs and makes it easy to integrate QU-Trade in an existing DBMS. In fact, a proposed method to index moving objects in Oracle can be seen as a special case of QU-Trade [7].

4. Generality. Since the index insertion, deletion, and query algorithms are used as black boxes, any proposal for their implementation can be used [19, 20, 24]. In addition, sophisticated main memory usage techniques [2] can be used with no modifications to the QU-Trade algorithms.

5. Online adaptation. There is no initial learning phase, and no assumptions about the update or query workload are made.

The rest of the paper is organized as follows. Section 2 covers related work. Section 3 covers QU-Trade, including the assumed architecture, the handling of modifications and queries, and policies for adjusting the sizes of the spatial extents. Section 4 presents an analytical study of the effect of window size, and Section 5 covers time-parameterization. Finally, Section 6 presents the empirical study, and Section 7 concludes and offers research directions.

2. RELATED WORK

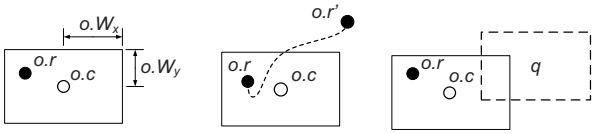
Several works have developed spatial and spatio-temporal indexes with low update cost. These indexes specify different points in the query-update space (Figure 2), and are thus suitable for a specific query-update mix. Some of them employ gridding and linearization with space-filling curves and use a B-tree [18, 29], possibly with periodic re-organization of the structure [5]. On the other hand, proposals based on the R-tree, notably the TPR-tree family [23, 24] have better query performance and are thus suitable for a different query-update mix [4]. We depart from this line of work by developing a mechanism that adapts to the workload using feedback from the updates as well as the queries. Not only is QU-Trade suitable for both low and high update-per-query rations, but it also adapts its performance to the incoming workload dynamically. Its update and query costs are not fixed at certain values as in traditional indexes, but vary according to the incoming workload. Combined, they minimize the average cost per operation. To the best of our knowledge, this problem has not been addressed before.

Some proposals revisit the R-tree insertion and deletion algorithms [19, 20] or employ smart use of main-memory caching [2] in order to speed up updates. These techniques are orthogonal to QU-Trade and can be used as the underlying index instead of a standard R*-tree.

Other proposals try to learn the mobility patterns of individual objects in order to reduce updates [6, 28]. These works can be seen as "update-aware" as the update cost is different for different objects, but are not workload-aware because the query and update costs are fixed, regardless of the incoming workload. Furthermore, they require an initial phase of learning from historical data, which QU-Trade does not.

When processing continuous spatio-temporal queries [11, 12, 16], prior knowledge about the registered queries can help in determining "safe regions" for objects. We address the complementary problem of answering snapshot queries, where no prior knowledge about queries is assumed.

Workload-awareness has appeared in three different settings. In approximate replication [22], queries are used to control the precision of remote sources. In model-driven data acquisition [9], a model is used to collect only the necessary data from remote sensors. Our approach is inspired by these works at an abstract level,



(a) Representation (b) Update (c) False positive

Figure 3: Object representation, update, and range search

albeit considering a very different problem. AGILE [10] proposes a workload-aware interval skip list, by moving objects to higher or lower levels of the list. The authors suggest that the same could be done for R-trees, but do not provide details. Due to the small height and high fanout of these indexes, this methodology is not attractive. In contrast, we achieve workload-awareness by controlling the filtering quality of the index. Our approach has the additional advantage that the implementation of the underlying index structure is not modified.

Online physical design tuning [3] also addresses the query-update tradeoff. However, the approach is to adaptively build and destroy an entire index as the workload changes. We consider the intermediate situation, in which an index always exists, but its filtering quality and hence its update and query performance are configured dynamically to adapt to the workload.

Finally, database cracking [17] addresses essentially the same problem in the context of a column store, albeit at the level of the database kernel. Spatio-temporal data sets are not considered.

3. QU-TRADE

Section 3.1 covers the underlying system architecture and Section 3.2 covers modifications. Then Sections 3.3 and Section 3.4 cover range, and incremental nearest-neighbor querying. Finally, Section 3.5 describes the window growing and shrinking mechanisms.

3.1 Representation and architecture

We assume a client-server architecture. The server keeps track of a population of moving objects, each of which has a unique object identifier oid. The objects continuously report their positions to the server, issuing an update *update*(oid, r'), where r' is a multidimensional point, denoting the new position of the object identified by oid. New objects can enter the system by issuing an insertion *insert*(oid, r), and objects can leave the system by issuing a delete *delete*(oid).

Users of the system issue range queries *range*(q_l, q_u) and nearest neighbor queries *nn*(q). We focus on snapshot queries, which are relevant to the applications that we are considering. Monitoring of continuous queries is a different problem where one can take advantage of prior knowledge of the registered queries [16]; we do not assume such knowledge in our setting. Rather, queries are ad-hoc and can occur anywhere at any time. Insertions, deletions, and updates, as well as range and NN queries make up the interface provided by the server.

The server-side representation of an object $o = (oid, r, c, W)$ consists of the object's identifier $o.oid$, its last reported position $o.r = (o.r_x, o.r_y)$, and a spatial window (c, W). The spatial window itself is represented by its center $o.c = (o.c_x, o.c_y)$ and its semi-width $o.W = (o.W_x, o.W_y)$, as shown in Figure 3(a).

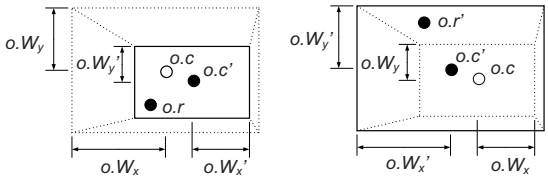
The server uses two data structures to support updates and queries. First, an index *HI* from object identifiers to records supports insertion, deletion, and retrieval of an object's record based on the object's identifier. We denote these operations as *HI.insert*(oid, o), *HI.delete*(oid), and *HI.retrieve*(oid). This index can be implemented as a hash index or a B-tree.

Second, a spatial index *SI* from the representations of object locations to object identifiers is assumed. We let this index be an R-tree. This index is built on the spatial windows of the objects, rather than their positions. The intuition is that if an object moves inside its window, its entry in the index does not necessarily have to be updated. Index *SI* supports insertion *SI.insert*$(oid, (c, W))$ and deletion *SI.delete*$(oid, (c, W))$, in addition to an intersection query *SI.intersect*(q_l, q_u) that returns the identifiers of the objects with windows that intersect with the query rectangle. This interface is quite general. Every spatial indexing method that supports indexing of rectangles and intersection queries can be used to answer range queries with QU-Trade. In order to support nearest neighbor search, we will later assume a hierarchical index akin to the R-tree.

3.2 Insertions, deletions, and updates

When an object enters the system and issues an insertion, it provides a new unique identifier *oid*. The server chooses an initial window semi-width $W_{initial}$ for the object, inserts its record into the database, inserts *oid* into *HI*, and inserts its window into *SI*.

When an object leaves the system and issues a deletion, the server deletes the object's record and issues deletions to each of the indexes *HI* and *SI*.



(a) Shrink operator (b) Grow operator

Figure 4: Shrinking and growing windows

When an object issues a positional update, the server retrieves the object's record using index *HI* and updates the object's position. The server then decides whether the update is to be propagated to spatial index *SI*. If so, the object's window is grown, yielding an enlarged window with a new window center (Figure 4(b)). Let P_g denote the probability of growing the window, and updating the spatial index. Pseudo-code for the update is given in Algorithm 1.

Algorithm 1 *update*(oid, r')

```
1: o ← HI.retrieve(oid)
2: o.r ← o.r'
3: with probability P_g do
4:     SI.delete(oid, (o.c, o.W))
5:     (o.c, o.W) ← grow(o.c, o.W, r')
6:     SI.insert(oid, (o.c, o.W))
```

Many alternative policies can be used to propagate the updates to the spatial index. In order to ensure that queries are answered correctly, we need to ensure that the object's last reported position always lies within the object's window. Thus, we demand that an update is always propagated if the newly reported position lies outside the object window:

$$P_g = 1 \text{ if } r' \notin [c - W, c + W]. \tag{1}$$

In the general case, the probability P_g with which the update occurs is a function of the window's width and center as well as the current and new positions of the object, $P_g = P_g(c, W, r, r')$. The simplest method propagates the update to the spatial index only when the new position lies outside the object window (as in Figure 3(b)).

$$P_g(c, W, r, r') = \begin{cases} 1 & \text{if } r' \notin [c - W, c + W], \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

We use this method in our implementation. However, alternative formulas for P_g (for example, a sigmoid function of the distance between the window center and the new position) can be used without modifying other parts of QU-Trade, as long as invariant 1 is satisfied.

3.3 Range queries

When a user issues a range query *range*(q_l, q_u) the server issues an intersection query to the spatial index *SI*. This returns a *candidate set* S of the identifiers of all the objects that can satisfy the query. All of these objects have windows that intersect the query window, and some of these objects may be *false positives* because although their windows intersect the query window, their positions may lie outside (see Figure 3(c)). These will be filtered out in an ensuing refinement phase. The constraint of equation 1 guarantees that there are no *false dismissals*. If an object lies within the query, its window will definitely intersect the query. This guarantees the correctness of the reported results. After the initial filtering step by the spatial index, the server invokes an internal *within*(oid, q_l, q_u) refinement operation for all the object identifiers in the candidate set. If this operation returns *false*, the identifier is dismissed from the candidate set.

Algorithm 2 *range*(q_l, q_u)

```
1: S ← SI.intersect(q_l, q_u)
2: for all oid ∈ S do
3:     if ¬within(oid, q_l, q_u) then
4:         S ← S − {oid}
5: return S
```

The within operator (Algorithm 3) retrieves an object's record using the index *HI*, and decides whether the object's position lies within the query window, returning *true* or *false* accordingly. In addition, with probability P_s, the object's window is shrunk resulting in a smaller window width and a new window center (Figure 4(a)). Then, an update with the new window is invoked on the spatial index *SI*.

Algorithm 3 *within*(oid, q_l, q_u)

```
1: o ← HI.retrieve(oid)
2: with probability P_s do
3:     SI.delete(oid, (o.c, o.W))
4:     (o.c, o.W) ← shrink(o.c, o.W, o.r, (q_l, q_h))
5:     SI.insert(oid, (o.c, o.W))
6: return o.r ∈ [q_l, q_u]
```

QU-Trade offers flexibility with respect to which updates to apply to the spatial index. In general, we would prefer to shrink and update the representations of the objects with very large windows that have large intersection areas with queries and that produce false positives.

Given a query, the probability of applying an update to an object is thus a function of the object's window and its relative position with respect to the query, $P_s = P_s(c, W, r, (q_l, q_h))$. A simple approach is to update the representation of all the false positives. The intuition is that if an object is a false positive, its window is probably too large and the object lies in a frequently queried area.

$$P_s(c, W, r, (q_l, q_u)) = \begin{cases} 1 & \text{if } r \notin [q_l, q_u], \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

We use this method in our implementation. More complex formulas for P_s (for example, an approach that takes into account the amount of overlap between the query and the object's window) can be used without modifications to QU-Trade.

3.4 Nearest-neighbor queries

We present how the standard incremental nearest neighbor algorithm [15] can be executed in the context of QU-Trade applied to an R-tree. The *INN*(q) algorithm (see Algorithm 4) maintains a priority queue, PQ, of entries and objects, and performs a best-first search in the index. The modification to the standard algorithm occurs because we now need to maintain a *candidate set* S of objects that have been inserted into the priority queue, but are not the nearest neighbor. This is analogous to the candidate set of false positives and true positives for range search. After the nearest neighbor has been found, some false positives are updated with a shrunk window in order to improve the quality of the spatial index.

Algorithm 4 *INN*(q)

```
1: nn ← null, S ← ∅, PQ ←empty priority queue
2: for all e ∈ SI.root do
3:     PQ.enqueue(⟨e, mindist(q, e)⟩)
4: repeat
5:     ⟨e, mindist(q, e)⟩ ← PQ.dequeue()
6:     if e is an object o then
7:         nn ← o
8:         exit the repeat loop
9:     else if e is a leaf entry then
10:        o ← HI.retrieve(e.oid)
11:        S ← S ∪ {o}
12:        PQ.enqueue(⟨o, dist(q, o)⟩)
13:    else if e is a non-leaf entry then
14:        n ← child of e
15:        for all e ∈ n do
16:            PQ.enqueue(⟨e, mindist(q, e)⟩)
17: until PQ ≠ ∅
18: S ← S − {nn}
19: for all o ∈ S do
20:    with probability P_s do
21:        SI.delete(o.oid, (o.c, o.W))
22:        (o.c, o.W) ← shrink(o.c, o.W, q)
23:        SI.insert(o.oid, (o.c, o.W))
```

Consider for example the situation depicted in Figure 5. While object o_2 is the nearest neighbor of q, object o_1 will be visited during search, yielding a false positive. Since the spatial index is built on the object windows, the search will have to retrieve the records of both objects in order to guarantee a correct answer. The false positive o_1 will be updated with a shrunk window, as shown in lines 19–23 of the *INN* algorithm. On the other hand, the object o_3 will be pruned during the search, as in the original incremental nearest neighbor algorithm.

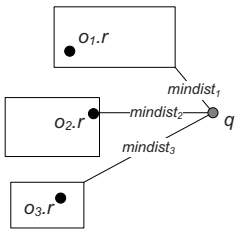


Figure 5: Nearest neighbor search

3.5 Growing and shrinking

QU-Trade provides both a mechanism for adaptation, and specific policies. The adaptation mechanism consists of growing and shrinking object windows when updating or querying an object respectively. This section describes our specific choices for the operators *shrink*(c, W, r, q) and *grow*(c, W, r'). Both operators return a new window for the argument object. The *grow* operator is called when an object update lies outside its current window. It will enlarge the window, using as additional information the newly reported position r'. The *shrink* operator is called upon a false hit

to a range or nearest neighbor query q. It will compress the window size, using as additional arguments the object's last reported position r and the query q.

In order to ensure the correctness of the reported query results, the operators have to satisfy the QU-Trade invariant; the last reported position of an object must never lie outside its window:

$$r' \in \mathit{grow}(c, W, r') \tag{4}$$

$$r \in \mathit{shrink}(c, W, r, q). \tag{5}$$

Further, the performance of QU-Trade depend on the specifics of these operators.

An important baseline configuration of QU-Trade is the one that never enlarges or compresses a window, and always keeps the windows centered at the object positions:

$$\mathit{shrink}_{\mathit{base}}(c, W, r, q) = (r, W) \tag{6}$$

$$\mathit{grow}_{\mathit{base}}(c, W, r') = (r', W). \tag{7}$$

In this case, all objects are bounded by windows of the same size W_{initial}. Determining the size of this initial window is a non-trivial optimization problem that depends on the mobility patterns of the moving objects. The resulting baseline QU-Trade configuration is equivalent to a proposal by Oracle to use the existing spatial indexing infrastructure of the Oracle server in order to index moving points [7]. Since the windows are always centered at the last reported position, the invariants 4 and 5 are trivially satisfied.

We consider additional policies for growing and shrinking. The simplest grow policy always keeps the window centered at an object's last reported position. In addition, it enlarges the window semi-width by a system-wide threshold parameter thr_g, while maintaining a sensible maximum bound, W_{max}:

$$\mathit{grow}_{\mathit{add}}(c, W, r') = (r', \min(W + thr_g, W_{max})). \tag{8}$$

Since the window is always centered at the last reported position, invariant 4 is trivially satisfied.

Although policy 8 can result in very efficient query and update processing, the threshold parameters it requires might be hard to set as they depend on the specifics of the application. For example, larger thresholds should be used for moving vehicles than pedestrians. Further, the thresholds are the same for all the objects, and do not account for the specifics of each object's motion. We have therefore derived a policy that estimates the center and radius of gyration [13] of an object's movement and then sets the window center and semi-width accordingly.

Let $\{r_1, \ldots, r_i\}$ be a sequence of reported positions of an object. The mean position is defined as $r_{cm} = \frac{1}{i}\sum_{j=1}^{i} r_j$, and the radius of gyration is defined as $\sqrt{R_g} = \sqrt{\frac{1}{i}\sum_{j=1}^{i}(r_j - r_{cm})^2}$. In real scenarios, objects move within a window centered at the mean position with a semi-width equal to their radius of gyration, which changes slowly over time when human movement is considered [13]. Thus, we incrementally estimate a discounted mean and radius of gyration of the object movement. When a positional update r' exceeds the current window $(r_{cm}, \sqrt{R_g})$, the mean and the square of the radius of gyration are updated as follows:

$$R_g = R_g + \gamma((r' - r_{cm})^2 - R_g), \gamma \in (0, 1)$$

$$r_{cm} = r_{cm} + \alpha(r' - r_{cm}), \alpha \in (0, 1).$$

The window is re-centered to the new mean, and its semi-width is enlarged by the square root of R_g:

$$\mathit{grow}_{\mathit{rg}}(c, W, r') = (r_{cm}, W + \sqrt{R_g}) \tag{9}$$

In order to initially satisfy invariant 4, the parameters α and γ have to satisfy the following inequality:

$$\gamma \geq \begin{cases} \alpha^2 & \text{if } \alpha \geq 0.5, \\ (1-\alpha)^2 & \text{if } \alpha \leq 0.5. \end{cases} \tag{10}$$

Essentially, the radius of gyration should grow faster than the mean in order to ensure that the window is large enough to accommodate the update. The proof, by simple algebraic manipulations, is omitted.

The discounting parameters α and γ control the impact of the motion history on the calculation of r_{cm} and R_g, respectively. The last reported position has a weight equal to α (γ), and the past motion history a weight equal to $1 - \alpha$ $(1 - \gamma)$ when calculating the new mean (radius of gyration). This acts as a forgetting mechanism when the object's movement characteristics change. This policy adapts to the object motion characteristics instead of using a fixed threshold for the window enlargement. It thus has a clear advantage over the growing policy 8, since its parameters are easier to set. Note that a method somewhat similar to the proposed mean-variance tree mechanism [28] can result as a special case of QU-Trade if the growing policy 9 is used and windows are never shrunk upon a false hit.

Two simple shrink policies always re-center the window to the object's last reported position. The first policy shrinks the window semi-width by a system-wide threshold parameter thr_s, while maintaining a sensible lower bound W_{min}. The second policy resets the window semi-width to the lower bound W_{min}:

$$\mathit{shrink}_{\mathit{sub}}(c, W, r, q) = (r, \max(W - thr_s, W_{min})) \tag{11}$$

$$\mathit{shrink}_{\mathit{reset}}(c, W, r, q) = (r, W_{min}). \tag{12}$$

The final shrinking policy finds the minimum distance between the query and the object's last reported position. It re-centers the window to the object's last reported position and then sets the window semi-width to the minimum distance:

$$\mathit{shrink}_{\mathit{je}}(c, W, r, q) = (r, \mathit{mindist}(r, q)). \tag{13}$$

Intuitively, this approach shrinks the window width "just enough," so that the object would not be a false positive for the same query. The invariant 5 is trivially satisfied in all cases, as the new window is always centered at the last reported position r.

4. THE EFFECT OF WINDOW SIZE

QU-Trade grows windows in order to avoid performing subsequent updates, thus reducing the update cost. The larger the window used for an object is, the more likely index updates for the object are to be shed. However, the cost of an individual update that is not shed also increases due to the increased overlap in the index. The challenge here is to study analytically whether the proposed practice of increasing the sizes of the windows can be beneficial in terms of leading to improved overall performance.

In order to address this challenge, we derive a simple probabilistic model of the effect of window size on the index's performance. Our analytical model consists of formulas for the query and update cost as a function of the average window size in the population. To keep the formulas tractable, we introduce several uniformity assumptions about the spatial and temporal distributions of updates and queries. These assumptions represent the worst case for QU-Trade, which is specifically designed to take advantage of skewed workloads. As a result, our analytical model will show that in the worst case, the cost benefits of shedding updates due to large windows overrules the overhead due to the increased cost of performing individual updates. Thus, even in the worst case, QU-Trade is

able to deliver performance benefits. In this section, we assume that an R-tree is used to index the object windows. We perform the analysis in 2-dimensional space. However, our results are straightforwardly generalizable to d-dimensional space.

We assume a population of N objects that move within the unit square, $[0,1]^2$. Queries of square shape and average width q in both dimensions appear uniformly. All the MBRs in the R-tree are square, and all the objects are bounded by square windows of average size W. Finally, we assume that the average fanout is the same at all levels. The height of the R-tree is $h = 1 + \lceil \log_f \frac{N}{f} \rceil \simeq \log_f N$. Under the above assumptions, we can use the following formula for the cost of an index traversal, C_{IT}, in an R-tree [27]: $C_{IT}(W,q) = 1 + \sum_{j=1}^{h-1} \frac{N}{f^j}(\sigma_j + q)^2$. The parameter σ_j is the average width of the squares at level j. For its computation, Theodoridis and Sellis [27] use the following argument. In N_j nodes with average size s_j^2 are grouped into N_{j+1} parent nodes with average size s_{j+1}^2 and each parent node groups f child nodes, the average size of a parent node is given by $s_{j+1} = (\sqrt{f} - 1)t_j + s_j$, where t_j is the average projected distance of two consecutive boxes. Its average value is given by $t_j = 1/\sqrt{N_j}$. This yields $\sigma_{j+1} = \sigma_j + \sqrt{f^j}\frac{\sqrt{f}-1}{\sqrt{N}}$ where $\sigma_0 = W$. The latter can be written in closed form as $\sigma_j = W + \frac{\sqrt{f}-1}{\sqrt{N}} \sum_{l=1}^{j} f^{l/2} = W + a_j$. This means that the cost of an R-tree traversal $C_{IT}(W,q)$ is a quadratic function of the average window size W as well as of the average query size q.

Let us disregard R-tree node splits and merges. The cost of an R-tree insertion C_{II} is constant with respect to the window size, as the insertion follows one path from the root to the leaf, choosing the best child at every level according to a heuristic (e.g., minimum area enlargement). Thus, $C_{II} = h$. The cost of a deletion C_{ID} of an object of width W is the cost retrieving that object, $C_{ID}(W) = C_{IT}(W,W)$. The cost of an index update is thus $C_{IU}(W) = h + C_{IT}(W,W)$.

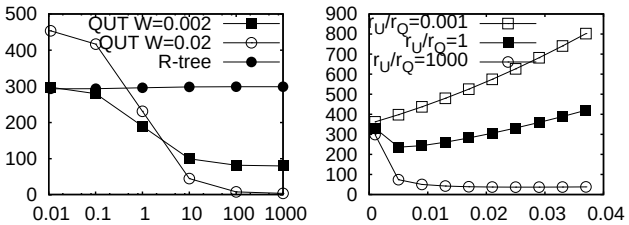
A QU-Trade update consists of two steps. First, the object's record is retrieved using the secondary index *HI*, upon which its position is updated. By using a main-memory hash index, the cost of this step is constant and equal to 2 I/O operations. Second, with probability P_g, the window is modified by the *grow* operator (this is done in the same time as Step 1, so it does not incur extra I/O), and an index update with the new window is invoked. The cost of the update is $C_U(W) = 2 + P_g(W)C_{IU}(W) = 2 + P_g(W)(h + C_{IT}(W,W))$.

The *within* operation also consists of two steps. First, the object's record is retrieved using the secondary index. The cost of this step is again constant and equal to 2 I/O operations. Second, with probability P_s, the window is modified by the *shrink* operator (at no extra I/O cost), and an index update with the new window is invoked. The cost of the *within* operation is thus $C_W(W,q) = 2 + P_s(W,q)C_{IU}(W) = 2 + P_s(W,q)(h + C_{IT}(W,W))$.

The range query consists of two steps. First, the tree is searched, and a candidate set is obtained. Second, the *within* operation is invoked for each item in the result. In order to derive the cost formula for range query, we need an estimate of the query selectivity. The probability that a window and a query intersect can be estimated as $(W+q)^2$, so the number of the candidate objects is $N(W+q)^2$ [27]. The cost of the range query is thus $C_Q(W,q) = C_{IT}(W,q) + N(W+q)^2 C_W(W,q)$.

Finally, the cost of a nearest neighbor search with argument q can be estimated using the cost of a range query with width equal to that of the so-called vicinity rectangle of q [25].

Let us assume a workload consisting of r_Q range queries per time unit and r_U updates per time unit. The average processing cost is then $C^{QUT}(W,q) = r_Q C_Q(W,q) + r_U C_U(W)$.



(a) Cost versus r_U/r_Q (b) Cost versus window size W

Figure 6: Cost model predictions

Figure 6 depicts the expectations of the cost model using a population of $N = 10^7$ objects and a fanout $f = 100$, which leads to an R-tree with 4 levels. The objects report their position whenever they have traveled a distance of $thr = 0.001$ (recall that movement occurs in the unit square).

Figure 6(a) depicts the average cost per operation while processing a fixed-size workload of queries and updates, $C^{QUT}(W,q)$, as a function of the updates-per-query ratio, r_U/r_Q. Note that the cost of QU-Trade is a function of r_U/r_Q, while the cost of the R-tree is nearly constant. This confirms that an R-tree defines a point in the query-update tradeoff space (Figure 2), whereas QU-Trade defines a curve in the same space. When the workload is dominated by queries (e.g., $r_U/r_Q = 0.01$), the cost of QU-Trade is close to that of an R-tree for small window sizes, and significantly larger for large window sizes (we depict the curves for $W = 2 \times thr, 20 \times thr$). When the workload is dominated by updates (e.g., $r_U/r_Q = 1000$), the cost of QU-Trade approaches zero. The situation here is the inverse, with large windows yielding a smaller average cost. It is clear that the best window size depends on the updates-per-query ratio of the workload. A growing and shrinking mechanism should yield small windows when r_U/r_Q is small (e.g., $W = 2 \times thr$), so that QU-Trade can be upper-bounded by the R-tree. It should yield large windows when r_U/r_Q is large (e.g., $W = 20 \times thr$), so that QU-Trade can achieve near-zero performance.

Figure 6(b) shows the effect of the window size W on the cost for different values of r_U/r_Q. When updates dominate the workload (e.g., $r_U/r_Q = 1000$), the cost is a monotonically decreasing function of W. When queries dominate the workload (e.g., $r_U/r_Q = 0.001$), the cost is a monotonically increasing function of W. This means that a window of zero area is the optimal choice for extremely query-heavy workloads and that a window of infinite width is the optimal choice for extremely update-heavy workloads. For workloads in-between these extremes (e.g., $r_U/r_Q = 1$), the cost is a convex function of W, meaning that an *optimal window* exists that minimizes the average cost. The query cost of QU-Trade, $C_Q(W,q)$, is an increasing function of W, whereas the update cost, $C_U(W)$, is a decreasing function of W. Their sum weighted by the query and update rates, which yields the average cost, becomes in general a convex function of W. This means that the increased cost of individual updates can be overruled by the benefit of discarding a number of updates. A growing and shrinking mechanism should enable convergence to the window size that minimizes the cost function of Figure 6(b), and it should adapt it as the workload's r_U/r_Q ratio changes. The experimental study in Section 6 presents three variants of QU-Trade that exhibit these capabilities.

5. TIME PARAMETERIZED QU-TRADE

In order to support predictive queries, an underlying index that

incorporates time can be used with QU-Trade. This assumes some form of prediction of the near-future movement. Essentially, any time-parameterized index that supports indexing of moving regions with extent is suitable for use with QU-Trade. In our implementation we use the TPR-tree [23].

The representation of an object $o = (oid, r(t), W(t))$ consists of the object's identifier, oid, its last reported position as a linear function of time, $r(t) = r_1 + v(t - t_1)$, and a time-parameterized rectangle, $W(t)$, termed the object's window. The latter can be represented by two time-parameterized points, $W(t) = (l(t) = l_0 + v^l(t - t_0), h(t) = h_0 + v^h(t - t_0))$. Figure 7(a) shows the one-dimensional case. Note that the velocities of the lower and upper bounds of a window, v_l and v_h are in general different, and may also differ from the last reported velocity v. The reference time t_0 of the window is the time of the last update of the object's entry in the index, which is in general different from the time of the last update of the object's entry in the database, t_1. It holds that $t_0 \leq t_1 \leq t_{now}$, where t_{now} denotes the current time. The object's actual movement (the dashed line in Figure 7(a)) remains within the window $W(t)$, until an index update occurs. As before, the spatio-temporal index is built on the objects' time-parameterized windows, rather than their positions.

When an object issues an update, it reports its new position r_2, velocity v_2, and reference time t_2. The object's record is updated via the index on object identifiers *HI*. The update is forwarded to the spatio-temporal index *STI* only if the new position is not predicted by the object's window: $r_2 \notin [l(t_2), h(t_2)]$. In that case, the object's window is grown via a *tp-grow* operator, $W'(t) =$ *tp-grow*$(W(t), t_2, r_2, v_2)$. Growing a time parameterized window consists of changing its reference time to t_2, growing the spatial extent of the window, and possibly widening the velocity bounding box, by enlarging v^h and lowering v^l (Algorithm 5).

Algorithm 5 *tp-grow*$(W(t), t_2, r_2, v_2)$

1: Let $W(t) = (l(t), h(t)) = (l_0 + v^l(t - t_0), h_0 + v^h(t - t_0))$
2: $c_0 \leftarrow (l_0 + h_0)/2$; $w_0 \leftarrow \|h_0 - l_0\|/2$
3: $(c'_0, w'_0) \leftarrow$ *grow*(c_0, w_0, r_2)
4: $l'_0 \leftarrow c'_0 - w'_0$; $h'_0 \leftarrow c'_0 + w'_0$
5: Update v^l, v^h
6: $l'(t) \leftarrow l'_0 + v^l(t - t_2)$; $h'(t) \leftarrow h'_0 + v^h(t - t_2)$
7: **return** $(l'(t), h'(t))$

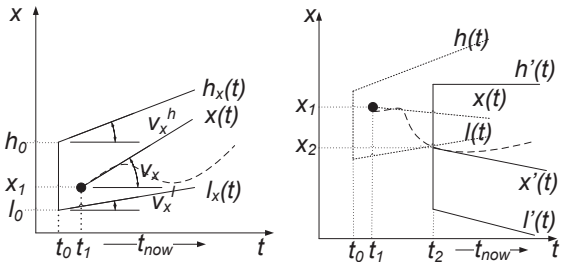
Figure 7(b) shows the one-dimensional case. The object moved out of its window $W(t) = (l(t), h(t))$ at time $t = t_2$. This is when the dashed line, representing the actual trajectory, crosses the dotted window $W(t)$. Thus, an update was forwarded to the index. The new object window $W'(t) = (l'(t), h'(t))$ has an enlarged width and an updated velocity bounding rectangle. Algorithm 6 shows the update algorithm, where

$$P_g((l(t), h(t)), t_2, r_2) = \begin{cases} 1 & \text{if } r_2 \notin [l(t_2), h(t_2)], \\ 0 & \text{otherwise.} \end{cases}$$

Algorithm 6 *update*(oid, t_2, r_2, v_2)

1: $o \leftarrow$ *HI.retrieve*(oid)
2: $o.r_0 \leftarrow o.r_2$; $o.t_0 \leftarrow o.t_2$
3: **with probability** P_g **do**
4: *STI.delete*$(oid, o.W(t))$
5: $o.W(t) \leftarrow$ *tp-grow*$(o.W(t), t_2, r_2, v_2)$
6: *STI.insert*$(oid, W(t))$

Timeslice, window, and moving window queries are supported [23].



(a) Representation (b) Update

Figure 7: TP-QU-Trade: Representation and index update.

In the general case, a window query is represented by a time parameterized rectangle $Q(t)$ and a time interval $[t_2, t_3]$. A window query is transformed to an "intersects" query and is issued to the spatio-temporal index *STI*. The index returns a candidate set of objects, whose windows intersect the query during the query interval. Each object in the candidate set is then examined, and the false positives are updated. An update to a false positive now involves shrinking its time-parameterized rectangle via a *tp-shrink* operator, $W'(t) =$ *tp-shrink*$(W(t), r(t), Q(t), [t_2, t_3])$. Shrinking a time-parameterized rectangle consists of changing its reference time, shrinking its spatial extent, and possible updating the window velocities v^l, v^h (Algorithm 7).

Algorithm 7 *tp-shrink*$(W(t), r(t), Q(t), [t_2, t_3])$

1: Let $r(t) = r_1 + v_1(t - t_1)$
2: Let $W(t) = (l(t), h(t)) = (l_0 + v^l(t - t_0), h_0 + v^h(t - t_0))$
3: Let $Q(t) = (q_l(t), q_h(t)) = (q_l + v_q^l(t - t_2), h_0 + v_q^h(t - t_2))$
4: $c_0 \leftarrow \frac{l(t_1)+h(t_1)}{2}$; $w_0 \leftarrow \frac{\|h(t_1)-l(t_1)\|}{2}$
5: $(c'_0, w'_0) \leftarrow$ *shrink*$(c_0, w_0, r_1, (q_l(t_2), q_h(t_2)))$
6: $l'_0 \leftarrow c'_0 - w'_0$; $h'_0 \leftarrow c'_0 + w'_0$
7: Update v^l, v^h
8: $l'(t) \leftarrow l'_0 + v^l(t - t_1)$; $h'(t) \leftarrow h'_0 + v^h(t - t_1)$
9: **return** $(l'(t), h'(t))$

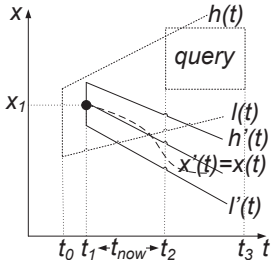


Figure 8: TP-QU-Trade: False positive.

Figure 8 shows a false positive result to a window query. Although the object's window $(l(t), h(t))$ intersects the query during the query time interval $[t_2, t_3]$, the object's last reported position $x(t) = x_1 + v(t - t_1)$ (which is recorded in the database) does not lie within the query. An update is propagated to the index with the updated window $W'(t) = (l'(t), h'(t))$. We omit detailed algorithms for the window queries, as their structure is similar to Algorithms 2 and 3.

Nearest-neighbor and reverse nearest-neighbor queries can also be supported, by extending the TPR-tree algorithms [1] in the same way as Algorithm 4. A best-first search in the tree is performed, and a candidate set is identified. Then, the false positives within the candidate set are updated, as in the window query case.

In this work, we have chosen not to update the window velocities in the time parameterized case (i.e., step 5 of Algorithm 5 and step 7 of Algorithm 7 are the identity functions). We leave dynamic

velocity bounds as future work. A cost analysis as in Section 4 can be performed for the TPR-tree case using the cost model of Tao et al. [24].

6. EXPERIMENTAL STUDY

We study three variants of QU-Trade. Section 6.1 details the settings. Section 6.2 presents a sensitivity analysis of the involved system parameters. Section 6.3 studies the query-update tradeoff, Section 6.4 studies the effects of spatial skew, and Section 6.5 studies scalability and the effects of varying query size. Section 6.6 studies the transient behavior of QU-Trade and the effect of temporal skew. Finally, Section 6.7 summarizes the results.

6.1 Experimental setup

We use an implementation of QU-Trade on top of an R*-tree and a variant of the TPR-tree that stores reference times in index entries. The spatial index library SaiL [14] forms the basis of our implementation.

All objects are initially assigned square windows of semi-width $W_{initial} = thr$. A positional update is propagated to the index only if it lies outside the object window as in Equation 2, and all false positives of a query are updated as in Equation 3.

We consider three variants of QU-Trade. The first, denoted *aggressive QU-Trade*, aggressively enlarges the window sizes, using the growing policy $grow_{add}$ of Equation 8 with parameters $thr_g = 20 \times thr$ and $W_{max} = 200 \times thr$. It also compresses window sizes aggressively, using the shrinking policy $shrink_{reset}$ of Equation 12 with $W_{min} = 0.1 \times thr$. The second variant, denoted *conservative QU-Trade*, enlarges windows slowly, using the growing policy $grow_{add}$ of Equation 8 with parameters $thr_g = 2 \times thr$ and $W_{max} = 20 \times thr$; it shrinks object windows slowly using the policy $shrink_{sub}$ of Equation 11 with parameters $thr_s = 2 \times thr$ and $W_{min} = thr$. The third variant, denoted *adaptive QU-Trade*, uses the growing policy $grow_{rg}$ of Equation 9 with parameters $\alpha = \gamma = 0.9$ and the shrink policy $shrink_{je}$ of Equation 13. These values stem from the sensitivity analysis presented in Section 6.2.

We compare QU-Trade with a stand-alone R*-tree, a solution with no index, and a grid-based approach. The R*-tree is a standard spatial index that is implemented in some database engines. It offers good query performance and is expected to behave well for query-intensive workloads. Grid-based approaches are common in continuous query monitoring for moving objects [11, 16]. The grid method is expected to perform well in update-intensive workloads. The number of grid cells, which has to be decided a priori, is a difficult parameter to set. A small cell size favors query performance, whereas a large cell size favors update performance. Since we intend to use the grid as a method with good update performance, we use a fairly sparse grid of 100 cells. For the time-parameterized case, we compare QU-Trade to a no-index approach, the TPR-tree, and the B^x-tree [18]. The B^x-tree has better update performance than the TPR-tree, and is thus expected to behave at its best in update-intensive workloads. In order to conduct a fair comparison, all the solutions use the same buffer, and the same index *HI* on the object identifiers, which is implemented as a main-memory hash table. Note that QU-Trade will also work with other implementations of *HI*.

We use synthetic workloads, as we want to experiment extensively with workload skew and the updates-per-query ratio. We have used a variation of the GSTD generator [26], as well as a network-based generator [23]. The objects update according to the point-based tracking policy, or the vector-based tracking policy in the time-parameterized case [8]. For the latter, the objects report their velocity along with their position. A threshold value thr is used for these policies. The parameters of the data generation are given in Table 1. The default values are shown in bold. Here u denotes the uniform distribution and N denotes a Gaussian distribution. We distinguish between directional movement, in which an object retains its speed between updates, and non-directional movement, in which an object changes its speed between updates.

Number of objects, N	**10K**, 100K, 1M
Page size	4096K
Fanout	100
Buffer size	$5\% \times N$
Fill factor	70%
TPR-tree horizon	100
Workspace width, S	$1000km$
thr	$S/200$
Initial position P	$\sim \mathbf{u}(\mathbf{0}, \mathbf{S}), \sim N(S/4, S/40)$
Velocity V	$\sim \mathbf{u}(-\mathbf{200km/h}, \mathbf{200km/h})$ $45km/h$, $90km/h$, $180km/h$ (network)
Query center C	$\sim \mathbf{u}(\mathbf{0}, \mathbf{S}), \sim N(S/2, S/20)$, $\sim N(S, S/40)$
Query extent E	$= \mathbf{S/80}, \sim u(S/100, S/80)$
Interval I between queries	$\sim u(S/400, S/40)$
Directional movement	**true**, false
Nodes in road network	20

Table 1: Index and workload parameters

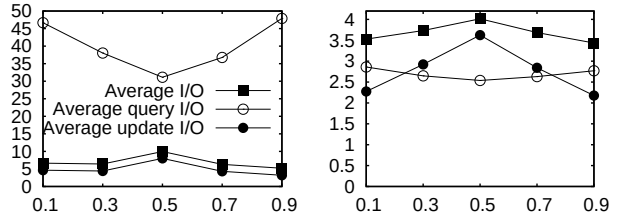
6.2 Sensitivity analysis of system parameters

First, we present a sensitivity analysis of the parameters involved in the growing and shrinking policies introduced.

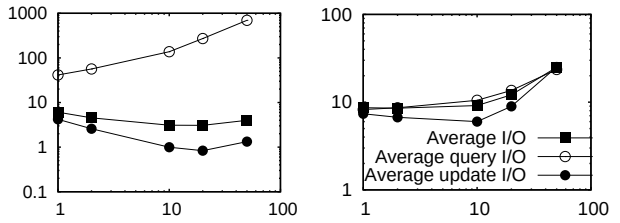
The growing policy $grow_{rg}$ of Equation 9 involves two parameters, α and γ, that control the impact of the current movement on the windows size. If α and γ are large, the current movement has higher weight than the past motion history. We vary α over $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. The corresponding γ values, due to the inequality of Equation 10, cannot be chosen freely. Their values were fixed at $\{0.9, 0.3, 0.5, 0.7, 0.9\}$, respectively. The average I/O per operation and the average I/O per update and query are shown in Figures 9(a) and 9(b) for an update-intensive (with update-to-query ratio $r_U/r_Q = 1000$) and a query-intensive (with $r_U/r_Q = 1$) workload, respectively[2]. We use the shrinking policy of Equation 12 with $W_{min} = thr$ in both cases. The extreme choices of α and the corresponding γ values minimize the update I/O, at the cost of extra query I/O. In contrast, the middle values maximize the update I/O and minimize the query I/O. We use the extreme values $\alpha = \gamma = 0.9$ for our subsequent experiments, as this minimizes the average I/O cost per operation in both cases. However, the I/O variation with respect to α and γ is very small, rendering the growing policy $grow_{rg}$ of Equation 9 robust to parameter variations.

Next, we consider the growing policy $grow_{add}$ of Equation 8. Figures 9(c) and 9(d) show the average I/O when varying the parameter thr_g for an update intensive, and a query-intensive workload, respectively (both axes are in logarithmic scale). The parameter W_{max} is fixed at $10 \times thr_g$, and the shrinking policy $shrink_{reset}$ of Equation 12 with $W_{min} = thr$ is used. In the update-intensive case, the value $thr_g = 20 \times thr_g$ minimizes the average I/O per operation. This value is chosen as the thr_g for the aggressive QU-Trade flavor. In the query-intensive case, the best value is $thr_g = 10 \times thr$. The query I/O is an increasing function of thr_g, whereas the update I/O is a convex function of thr_g. When windows become too large, the cost to perform an individual update increases due to the increased overlap. However, for small enough values of thr_g, the update cost is a decreasing function of thr_g. This leads us to the choice of $thr_g = 2 \times thr$ for the conservative

[2]The experimental graphs do not have their axes labeled. All the measured and varied quantities are listed in the figure captions.

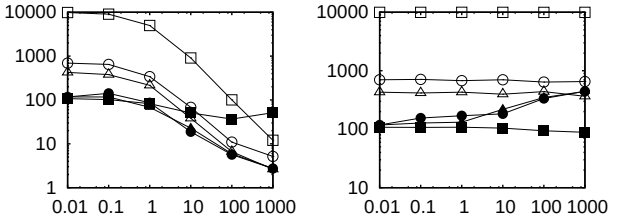


(a) I/O versus α, $r_U/r_Q = 1000$ (b) I/O versus α, $r_U/r_Q = 1$

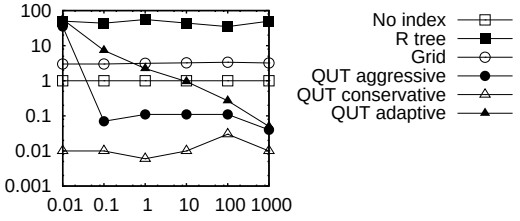


(c) I/O versus thr_g/thr, $r_U/r_Q = 1000$ (d) I/O versus thr_g/thr, $r_U/r_Q = 1$

Figure 9: Sensitivity to grow parameters



(a) Average I/O versus r_U/r_Q (b) Average query I/O versus r_U/r_Q



(c) Average update I/O versus r_U/r_Q

Figure 10: Query-update tradeoff

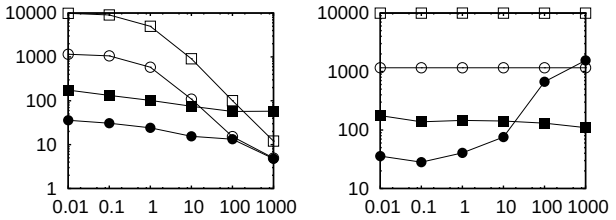
QU-Trade.

The parameters chosen for the shrinking policies stem from a similar analysis. We omit the details due to lack of space.

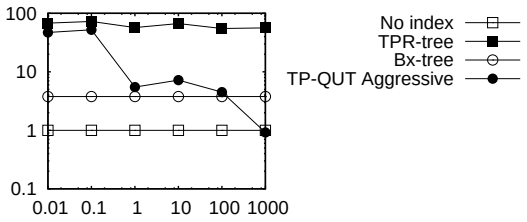
6.3 The query-update tradeoff

The most important characteristic of QU-Trade is adaptivity with respect to the updates-per-query ratio, r_U/r_Q. Our first experiments confirm that QU-Trade indeed is capable of achieving this adaptivity. We use the network-based data generator for the update workload and a uniform range query workload. We vary r_U/r_Q from 0.01 to 1000. Figure 10(a) shows the average I/O cost per operation (update or query) for the spatial case, and Figure 11(a) shows the same metric for the time-parameterized case. All axes are in logarithmic scale.

First, note that using an R-tree or a TPR-tree is not beneficial in an update-intensive environment in terms of average cost per operation. The cost of maintaining these indexes in the presence of updates exceeds the performance gains for the queries. In our setting, when r_U/r_Q exceeds 100, a solution that does not index the object locations at all achieves better average cost per operation. However, in query-intensive settings, using such an index results in



(a) Average I/O versus r_U/r_Q (b) Average query I/O versus r_U/r_Q



(c) Average update I/O versus r_U/r_Q

Figure 11: Query-update tradeoff, time-parameterized

two orders of magnitude better performance. Thus, it is of interest to achieve adaptivity with respect to r_U/r_Q.

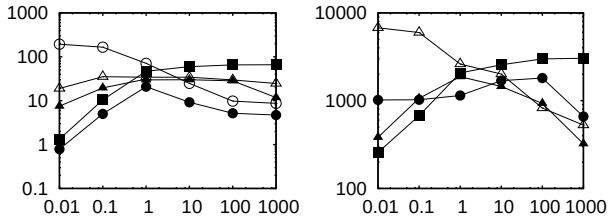
We start with the spatial case (Figure 10(a)). The average costs per operation of the adaptive and aggressive QU-Trade are upper-bounded by the average cost of the R-tree in query-intensive workloads ($r_U/r_Q \in [0.01, 1]$). When the workload becomes update-intensive, their average costs drop to a value close to 1 I/O.

This behavior can be explained by examining the average I/O cost per query operation (Figure 10(b)) and the average I/O cost per update operation (Figure 10(c)). The average query cost rises as the workload becomes update-heavy, ranging from the query cost of the R-tree up to the query cost of the grid index. Conversely, the update cost falls from the update cost of the R-tree down to zero as the workload becomes update-intensive. The combination of the query and update cost result in the desired adaptivity. The conservative QU-Trade achieves similar costs to the grid index in this setting. Both methods perform well in update-intensive workloads, but their costs are too high in the query-intensive case. This is explained by their high query costs (Figure 10(b)). In the next section, we show that the presence of spatial skew enables the conservative variant to achieve the desired adaptivity and to achieve similar performance as the aggressive and the adaptive variants. Note that the query cost of QU-Trade never exceeds that of the grid index.

In the time-parameterized case (Figures 11(a), 11(b), and 11(c)), the situation is similar (only the aggressive policy is depicted). QU-Trade's update cost drops, and its query cost rises as the workload becomes update-intensive. This enables its average cost per operation to be upper bounded by the TPR-tree's cost and its overhead to be minimal in update-intensive workloads. The query cost is never higher than that of the B^x-tree. In subsequent experiments, we restrict ourselves to the non time-parameterized case of QU-Trade operating on an R-tree. These experiments have shown that QU-Trade effectively covers the space between maintaining an accurate R-tree or TPR-tree, and having no index at all. Query latency is not sacrificed, as it is never higher than that of an update-efficient index (grid or B^x-tree).

Figure 12(a) shows the average I/O for the same experimental setup, but for nearest-neighbor queries (the legend of Figure 10 is assumed, both axes are in logarithmic scale, and the case with no index is not depicted). Similar observations can be made for this case. For our subsequent experiments, we restrict ourselves to range queries.

Finally, we study the performance of QU-Trade when the index



(a) Average I/O versus r_U/r_Q, NN queries (b) Average time ($\mu secs$) versus r_U/r_Q, main memory execution

Figure 12: Query-update tradeoff, NN queries and main memory indexing

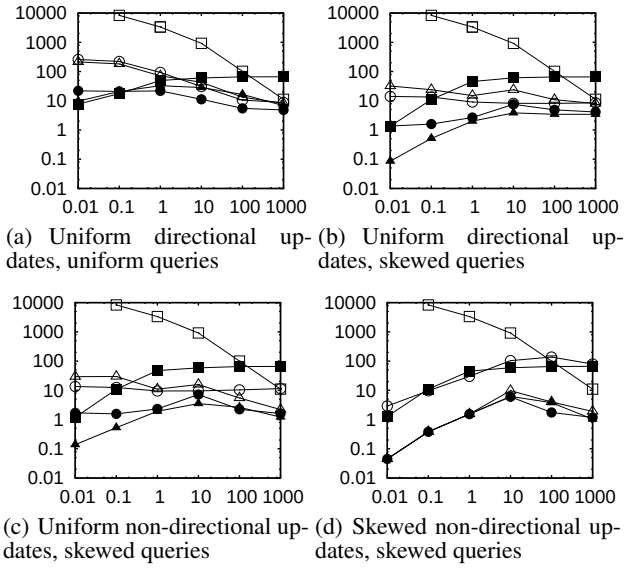
it controls resides in main memory rather than on disk. We perform the same experiment, changing the underlying storage manager to a main memory one. We do not optimize the indexes for main memory (e.g., changing the R-tree node size). Figure 12(b) shows the average execution time per operation for all the methods in $\mu secs$ (the legend of Figure 10 is assumed, both axes are in logarithmic scale, and the case with no index is not depicted). The R-tree cost rises as the workload becomes update-intensive. The cost of the conservative QU-Trade drops as the workload becomes update-intensive. In contrast, the average execution cost of the aggressive, and especially the adaptive QU-Trade, is a concave curve. The cost is minimized at the extremes, due to the good update behavior for update-intensive workloads and the good query behavior for query-intensive workloads. We conclude that the query-update tradeoff is inherent, both in buffered, disk-resident indexes and in indexes that lie entirely in main memory. QU-Trade can address this tradeoff in both settings.

6.4 The effect of spatial skew

QU-Trade benefits from spatial skew. Our next experiments show that the QU-Trade curve in the query-update tradeoff space (Figure 2) moves towards optimal performance as the spatial skew increases.

We consider the query-update tradeoff for four workloads with increasing spatial skew. The first workload consists of uniform directional object updates and uniform queries. The second workload introduces query skew, by using a Gaussian spatial query distribution with the mean at the center of the work space. The third workload changes the nature of the object movement. The objects now move non-directionally, by choosing a random velocity vector at every update. This results in a random walk, rather than a linear movement. As before, the queries follow a Gaussian distribution. The fourth workload, introduces a Gaussian initial spatial distribution and a Gaussian query distribution concentrated at the top-right corner of the working space. The average I/O performance for the three latter workloads is shown in Figures 13(a), 13(b), 13(c), and 13(d), respectively (the legend of Figure 10 is assumed and both axes are in logarithmic scale).

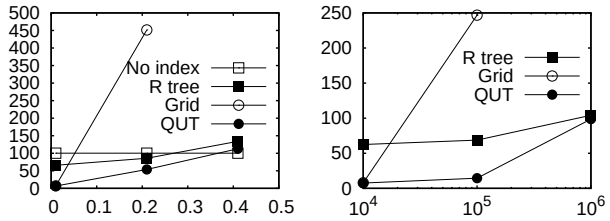
A comparison between Figure 13(b) with the uniform workload in Figure 13(a) indicates that especially the adaptive QU-Trade benefits a lot from the concentrated nature of the query distribution. This allows updates to be dropped even when the workload is query-intensive. Its performance is the best among all solutions, including the R-tree, for query-intensive as well as update-intensive workloads. Moving to increasing levels of skew in Figure 13(c) and Figure 13(d), the conservative solution approaches the aggressive and adaptive solutions. The spatial skew now renders the conservative variant workload-aware. The grid index suffers from the skew and performs worse than even the R-tree for high update rates. Using a structure that relies on uniformity assumptions, such as the grid, fails when faced with substantial skew. In all cases, QU-Trade



(a) Uniform directional updates, uniform queries (b) Uniform directional updates, skewed queries

(c) Uniform non-directional updates, skewed queries (d) Skewed non-directional updates, skewed queries

Figure 13: Workload skew, average I/O versus r_U/r_Q



(a) Average I/O versus query size, $r_U/r_Q = 100$ (b) Average I/O versus N

Figure 14: Query effect, and scalability

variants perform increasingly better as the spatial skew increases. The aggressive and adaptive variants consistently exhibit the best performance, followed by the conservative approach.

6.5 The effect of queried space and scalability

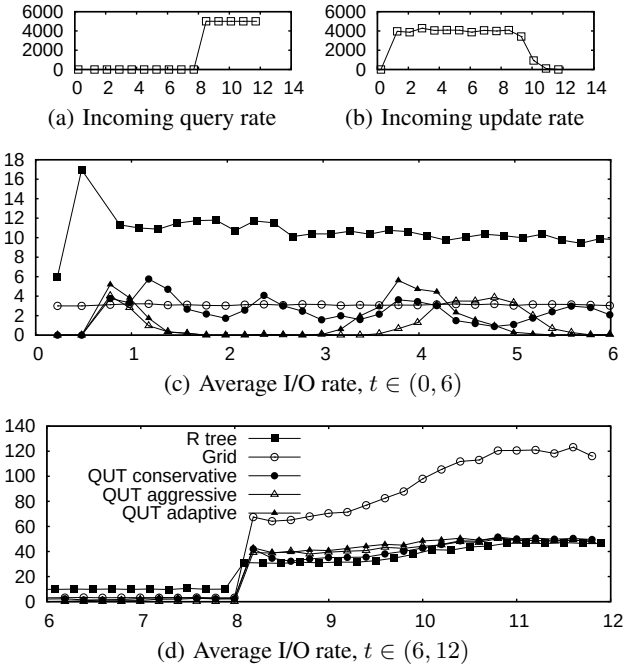
Since QU-Trade is built on an R-tree or a TPR-tree, it inherits the behavior of these indexes with respect to query size. Figure 14(a) depicts the effect of the query area on the average performance. The workload is uniform as in Section 6.3. While the grid solution suffers from large queried regions, the R-tree as well as QU-Trade are not affected substantially, although they exhibit quadratic behavior. The behavior is similar for all the QU-Trade variants.

Figure 14(b) shows that QU-Trade also inherits the scalability of the R-tree, whereas the grid proves non-scalable when the number of grid cells remains constant. The behavior is similar for all the QU-Trade variants.

6.6 Transient behavior

We move to study the temporal evolution of performance as the characteristics of a workload change over time. We are interested in both stationary workloads that intermix queries with updates and workloads that exhibit temporal skew. In the latter workloads, a high number of updates with relatively few queries is followed by a query burst. Due to lack of space, we present our results for the non-stationary cases only.

In the subsequent experiments, we measure the average I/O rate as it evolves over time. The average I/O rate measures how many I/Os per operation and time unit occur in order for a given workload to be processed. A low average I/O rate means that a workload can be processed efficiently. Note that the performance differences between the various methods naturally seem smaller with this metric,

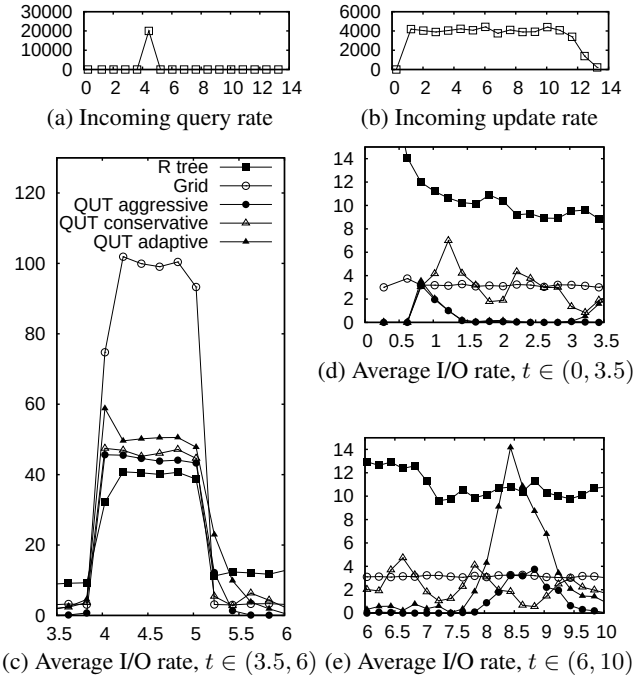


(a) Incoming query rate (b) Incoming update rate

(c) Average I/O rate, $t \in (0, 6)$

(d) Average I/O rate, $t \in (6, 12)$

Figure 15: Transient response, queries after updates



(a) Incoming query rate (b) Incoming update rate

(c) Average I/O rate, $t \in (3.5, 6)$

(d) Average I/O rate, $t \in (0, 3.5)$

(e) Average I/O rate, $t \in (6, 10)$

Figure 16: Transient response, queries between updates

as I/O operations are now averaged over a very small time period, and not across the whole workload.

First, we consider a scenario where an update-only workload from $t = 0$ to $t = 8$ dropping to zero updates at $t = 10$ (Figure 15(b)) is combined with a query-only workload from $t = 8$ to $t = 12$ (Figure 15(a)). Figures 15(c) and 15(d) show the evolution of the average I/O rate of the five compared methods for $t \in (0, 6)$ and $t \in (6, 12)$, respectively. Figure 15(c) shows that the QU-Trade solutions are roughly upper-bounded by the grid index in the update-heavy part of the workload, with the R-tree having much worse update performance. Conversely, in the query-heavy part (Figure 15(d)), the QU-Trade solutions exhibit performance similar to that of the R-tree, while the grid exhibits poor query performance. The QU-Trade techniques can thus combine minimal update performance during the update-intensive part, and quickly move to a query performance close to that of the R-tree when the queries arrive.

For our final experiment, we use a constant-rate update workload (Figure 16(b)) intermixed with a very high rate query burst, roughly in the time interval $(3.5, 5)$, as shown in Figure 16(a). We want to determine the overhead of updating the false positives found in the queries after a series of updates have enlarged the object windows, and how quickly the windows will be enlarged again after the query burst. Figure 16(d) shows the average I/O rate before the query burst ($t \in (0, 3.5)$). The aggressive and the adaptive QU-Trade variants have near-zero overhead. The grid and the conservative QU-Trade have an overhead of 3 I/Os per time unit on average, and the R-tree an update overhead of 10 I/Os per time unit. When the queries arrive (Figure 16(c), $t \in (3.5, 6)$), the grid index exhibits the steepest increase in I/O rate and exceeds by far the R-tree in terms of average I/O rate. The R-tree exhibits the smoothest slope. The QU-Trade techniques increase their I/O rate to about the same level as the R-tree. This means all QU-Trade variations manage to shrink the window sizes quickly. When the queries are over (Figure 16(e), $t \in (6, 10)$), the aggressive QU-Trade quickly returns to its near-zero update overhead, since it enlarges the window sizes quickly. The adaptive and conservative variations exhibit oscillations before they return to their stable behavior.

6.7 Summary

We have studied the average and temporal behavior of three variants of QU-Trade in environments with disk-based and main-memory indexes, uniform and skewed workloads, and spatial and spatio-temporal indexes. Here, we briefly summarize the most important findings.

First, our experiments indicate that workload-awareness is desirable in a practical system. For query-intensive workloads with few updates, the average performance of an update-efficient index (grid or B^x-tree) is less than an order of magnitude better than the naive approach (no indexing). In contrast, a query-efficient index (TPR-tree or R-tree) can achieve two or more orders of magnitude better performance than the naive approach. However, in update-intensive workloads with few queries, R- and TPR-trees incur large update overheads, resulting in an order of magnitude worse performance than the naive approach. The query and update frequencies of real workloads fluctuate greatly over time. QU-Trade achieves the best of both worlds by adapting its behavior to the incoming workload. Its performance for query-intensive workloads is two or more orders of magnitude better than the naive approach, and its performance is nearly optimal in update-intensive workloads. It is thus a very practical solution for real-world scenarios.

Second, spatial skew has been shown to be beneficial for QU-Trade. Its performance (across any workload, query- or update-intensive) becomes better by two orders of magnitude when we introduced substantial skew in the workload. Thus, QU-Trade is expected to deliver its best performance in real workloads.

Third, QU-Trade has been shown to have a smooth transient behavior. When the workload's r_U/r_Q changes dynamically, QU-Trade follows these changes fast, avoiding temporary performance deterioration.

Finally, the aggressive QU-Trade variant exhibits the best performance overall. Interestingly, it has the best transient behavior, oscillating less when the incoming workload changes rapidly. Thus, it is desirable to be aggressive when growing and shrinking windows.

However, we recommend the adaptive variant as the most practical because it is practically parameterless and its performance is very close to that of the aggressive variant. The adaptive QU-Trade variant has only two parameters, α and γ that control the impact of past motion history. However, its performance is very robust with respect to parameter variations, staying within the same order of magnitude across the whole spectrum of values for α and γ.

7. CONCLUSIONS AND FUTURE WORK

QU-Trade is a framework equipped with specific techniques that render R-trees and TPR-trees workload-aware, and as a result covers the space in-between maintaining an accurate index and having no index at all. Moving object positions are represented by rectangles, the sizes of which are maintained dynamically in order to balance query and index update performance. Objects that update frequently, but move in areas with few queries, obtain large windows that yield fewer index updates. When an object is a false positive in a query, its window is shrunk, as it has entered a frequently queried region.

An analytical model and an extensive experimental study offer insights into the worst case, the average, and the transient behavior of three variants of QU-Trade. The studies show that QU-Trade is capable of reducing the update overhead of the R-tree and the TPR-tree to a near-zero cost while maintaining good query performance. QU-Trade can outperform these indexes by up to two orders of magnitude for update-heavy workloads, while achieving virtually comparable performance for query-heavy workloads. Unlike grid-based indexes, QU-Trade thrives on workloads with spatial skew. Further, QU-Trade is capable of quickly adapting to workloads that change over time. QU-Trade is a general and versatile solution. It is applicable to a variety of indexes in the R-tree family, and is easily deployable in existing database engines. It makes no assumptions about a workload's spatial and temporal characteristics, but rather adapts to those in a fully online manner. We therefore view QU-Trade as a practical way to support moving object indexing in an industrial-strength DBMS.

This research opens several lines of work that we plan to pursue. First, we would like to experiment with changing the velocity bounding boxes of the object windows when growing and shrinking. Second, QU-Trade may be applied to other R-tree type indexes for use with sensor network workloads. Finally, our current techniques for growing and shrinking object windows are heuristic. We would like to take a more principled approach and investigate what an optimal solution to this problem means, and under what assumptions optimality can be meaningful.

Acknowledgements

The authors would like to thank Marios Hadjieleftheriou for making the SaiL library available under the GNU LGPL. This research was funded in part by grant 645-06-0517 from the Danish Agency for Science, Technology and Innovation. It was conducted in part while C. S. Jensen was a visiting scientist at Google Inc.

8. REFERENCES

[1] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Šaltenis. Nearest and reverse nearest neighbor queries for moving objects. VLDB J., 15(3):229–249, 2006.

[2] L. Biveinis, S. Šaltenis, and C. S. Jensen. Main-memory operation buffering for efficient R-tree update. In *VLDB*, pp. 591–602, 2007.

[3] N. Bruno, and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, pp. 826–835, 2007.

[4] S. Chen, C. S. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. In *VLDB*, pp. 1574–1585, 2008.

[5] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento. ST^{2}B-tree: a self-tunable spatio-temporal B$^+$-tree index for moving objects. In *SIGMOD*, pp. 29–42, 2008.

[6] R. Cheng, Y. Xia, S. Prabhakar, and R. Shah. Change tolerant indexing for constantly evolving data. In *ICDE*, pp. 391–402, 2005.

[7] R. K. V. Kothuri, B. Hanckel, and A. Yalamanchi. Using Oracle extensibility framework for supporting temporal and spatio-temporal applications. In *TIME*, pp. 15–18, 2008.

[8] A. Civilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE Trans. Knowl. Data Eng.*, 17(5):698–712, 2005.

[9] A. Deshpande, C. Guestrin, S. Madden, and J. M. Hellerstein. Model-driven data acquisition in sensor networks. In *VLDB*, pp. 588–599, 2004.

[10] J.-P. Dittrich, P. M. Fisher, and D. Kossmann. AGILE: Adaptive indexing for context-aware information filters. In *SIGMOD*, pp. 215–226, 2005.

[11] B. Gedik, L. Liu, K.-L. Wu, and P. S. Yu. LIRA: Lightweight, region-aware load shedding in mobile CQ systems. In *ICDE*, pp. 286–295, 2007.

[12] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. MobiQual: QoS-aware load shedding in mobile CQ systems. In *ICDE*, pp. 1121–1130, 2008.

[13] M. C. González, C. A. Hidalgo, and A.-L. Barabási. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, 2008.

[14] M. Hadjieleftheriou, E. G. Hoel, and V. J. Tsotras. SaiL: A spatial index library for efficient application integration. *GeoInformatica*, 9(4):367–389, 2005.

[15] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[16] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, pp. 479–490, 2005.

[17] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD Conference*, pp. 413–424, 2007.

[18] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B$^+$-tree based indexing of moving objects. In *VLDB*, pp. 768–779, 2004.

[19] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update R-tree. In *MDM*, pp. 113–120, 2002.

[20] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: A bottom-up approach. In *VLDB*, pp. 608–619, 2003.

[21] O. A. Nielsen, G. Jovicic. The AKTA road pricing experiment in Copenhagen In *10th Int. Conf. on Travel Behaviour Research*, 2003.

[22] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *SIGMOD*, pp. 355–366, 2001.

[23] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pp. 331–342, 2000.

[24] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pp. 790–801, 2003.

[25] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE Trans. Knowl. Data Eng.*, 16(10):1169–1184, 2004.

[26] Y. Theodoridis, and M. A. Nascimento Generating spatiotemporal datasets on the WWW. *SIGMOD Rec.*, 29(3):39–43, 2000.

[27] Y. Theodoridis, and T. K. Sellis. A model for the prediction of R-tree performance. In *PODS*, pp. 161–171, 1996.

[28] Y. Xia, S. Prabhakar, S. Lei, R. Cheng, and R. Shah. Indexing continuously changing data with mean-variance tree. In *SAC*, pp. 1125–1132, 2005.

[29] M. L. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *VLDB J.*, 17(3):379–400, 2008.