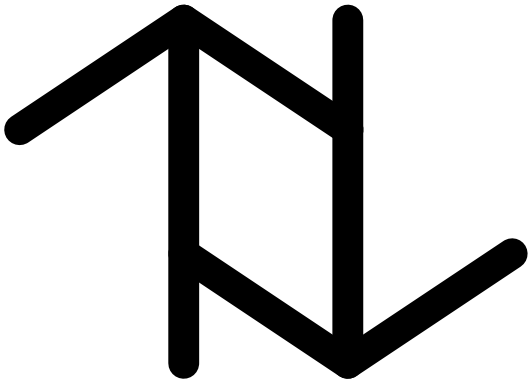

The COST Benchmark—Comparison and Evaluation of Spatio-Temporal Indexes

Christian S. Jensen and Dalia Tiešytė and Nerius Tradišauskas

July 17, 2006

TR-86

A TIMECENTER Technical Report

Title	The COST Benchmark—Comparison and Evaluation of Spatio-Temporal Indexes
Author(s)	Christian S. Jensen and Dalia Tiešytė and Nerius Tradišauskas
Publication History	July 2006. A TIMECENTER Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Faiz A. Currim, Sabah A. Currim, Bongki Moon, Sudha Ram, Stanley Yao

Individual participants
Yun Ae Ahn, Chungbuk National University, Korea; Michael H. Böhlen, Free University of Bolzano, Italy; Curtis E. Dyreson, Washington State University, USA; Dengfeng Gao, Indiana University South Bend, USA; Fabio Grandi, University of Bologna, Italy; Heidi Gregersen, Aarhus School of Business, Denmark; Vijay Khatri, Indiana University, USA; Nick Kline, Microsoft, USA; Gerhard Knolmayer, University of Bern, Switzerland; Carme Martín, Technical University of Catalonia, Spain; Thomas Myrach, University of Bern, Switzerland; Kwang W. Nam, Chungbuk National University, Korea; Mario A. Nascimento, University of Alberta, Canada; John F. Roddick, Flinders University, Australia; Keun H. Ryu, Chungbuk National University, Korea; Dennis Shasha, New York University, USA; Michael D. Soo, amazon.com, USA; Andreas Steiner, TimeConsult, Switzerland; Paolo Terenziani, University of Torino, Italy; Vassilis Tsotras, University of California, Riverside, USA; Fusheng Wang, Siemens, USA; Jef Wijsen, University of Mons-Hainaut, Belgium; and Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.aau.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

An infrastructure is emerging that enables the positioning of populations of on-line, mobile service users. In step with this, research in the management of moving objects has attracted substantial attention. In particular, quite a few proposals now exist for the indexing of moving objects, and more are underway. As a result, there is an increasing need for an independent benchmark for spatio-temporal indexes.

This report characterizes the spatio-temporal indexing problem and proposes a benchmark for the performance evaluation and comparison of spatio-temporal indexes. Notably, the benchmark takes into account that the available positions of the moving objects are inaccurate, an aspect largely ignored in previous indexing research. The concepts of data and query enlargement are introduced for addressing inaccuracy. As proof of concepts of the benchmark, the report covers the application of the benchmark to three spatio-temporal indexes—the TPR-, TPR*-, and B^x-trees. Based on conceptual analyses of the indexes, performance hypotheses are formulated. Experimental results and consequent guidelines for the usage of these indexes are reported.

1 Introduction

With the availability of mobile computing technologies, geo-positioning, and wireless communication capabilities, it has become possible to accumulate the changing locations of populations of moving objects in real time. Consumer electronics are affordable, current Global Positioning System (GPS) [1] receivers are capable of geo-positioning with an accuracy of up to a few meters, the General Packet Radio Service (GPRS) [2] and similar technologies have become common and relatively cheap means of wireless data transfer. It is thus possible for an object to continually obtain and transmit its current position to a central server.

Applications are emerging that require or may benefit from the tracking of the locations of moving objects. These occur in areas such as logistics, traffic management, public transportation, and location-based services. Current applications usually track only relatively small numbers of objects, but as the underlying technologies continue to improve, applications that concern large numbers of objects are on the horizon.

The increasing interest in mobile location data has served as motivation for the development of spatio-temporal indexes for the current and near-future positions of moving objects. A number of spatio-temporal indexes have been proposed, such as R-tree-based indexes, e.g., the TPR-tree [3], the TPR*-tree [4], the STAR-tree [5], and the R^{EXP}-tree [6]; the quadtree-based index STRIPES [7], and the B^+-tree-based B^x-tree [8], to name but a few.

This continuing proliferation of indexing techniques creates a need for a standard procedure for performance evaluation and comparison. Although mathematical complexity analysis is valuable, empirical evaluation [9] is indispensable for evaluation and comparison of spatio-temporal indexing techniques. The current state of affairs is that indexes being proposed are being evaluated empirically and are being compared to, typically, one other indexing technique. The empirical studies reported are rarely exhaustive and, not surprisingly, tend to focus on the favorable qualities of the index being proposed. The availability of an independent benchmark specification establishes an equal footing for obtaining experimental results and enables broader comparison.

This report proposes a benchmark specification, termed COST, for the evaluation and comparison of spatio-temporal indexes. The benchmark is independent in the sense that it is proposed independently of a specific indexing technique. The benchmark aims to provide a unified procedure that covers an extensive variety of possible and realistic settings. In particular, the benchmark evaluates the index ability to accommodate uncertain object positions. Queries and updates are considered, as are both I/O and CPU performance.

The remainder of this report is outlined as follows. Related work is covered in Section 2. The addressed indexing problem is detailed in Section 3. Sections 4 and 5 contain the benchmark specification. Section 6 introduces to spatio-temporal indexes, performs conceptual analysis of their performance, and reports on experimental results that were obtained using the benchmark. Section 7 concludes and offers directions of future work.

2 Related Work

We cover in turn existing benchmarks for spatio-temporal data, previous work on the indexing of uncertain data, and past empirical evaluations of spatio-temporal indexes.

A number of benchmarks exist that measure transaction performance in traditional database systems. For example, a set of benchmarks that evaluate system performance and price is provided by Gray [10]. However, these benchmarks are not applicable to spatio-temporal data.

Of relevance to moving objects, Theodoridis [11] provides a benchmark that includes a database description and 10 non-predictive queries for the static and moving spatial data. Myllymaki and Kaufman [12] also propose a benchmark for moving objects. The query and update performance measure is CPU time, as a main-memory resident index is assumed. Future queries on anticipated future locations are not considered. Werstein [13] proposes a benchmark for 3-dimensional spatio-temporal data. The benchmark is oriented towards general operating system and database system performance comparison, including evaluation of the spatio-temporal and 3-dimensional capabilities. Zobel et al. [9] provide general guidelines for the comparison of indexing techniques. The authors list criteria by which the indexing techniques should be compared and give four comparison methodologies. Tzouramanis et al. [14] perform an extensive, rigorous experimental comparison of four types of quadtree-based spatio-temporal indexes, using the same benchmark specification when performing experiments with the four indexes. Their proposal concerns raster data, generated with the G-TERD benchmark tool.

The concept of data uncertainty for moving object positions has previously been studied quite extensively (see, e.g., [15, 17, 18, 19]). While the bulk of this work has been conducted independently of indexing, some works (see, e.g., [16, 19]) offer insights into the indexing of uncertain positions. The present work goes further by proposing a simple and yet effective method for storing and retrieving position data with accuracy guarantees. Existing indexes can straightforwardly be extended to accommodate such data.

Many authors of spatio-temporal indexes have compared their indexes to usually one other competitive index (e.g., [3, 4, 7, 8]). However, these comparisons tend to focus on exploring the properties of the new index being proposed; and with the new index being the main topic, the experimental specifications are relatively limited and lack independence.

The benchmarks covered above consider neither uncertain data nor accuracy guarantees. DynaMark [12] shares similarities with the COST benchmark with respect to the generated traffic data, but it ignores aspects to do with future positions. To the best of our knowledge, no independent benchmark exists that has been designed specifically for the evaluation of disk-based indexes for the current and near-future uncertain positions of moving objects.

3 Spatio-Temporal Indexing

This research is concerned with the indexing of large amounts of current and near-future, 2-dimensional moving object positions, and predictive spatial queries are of interest. In this setting, position data are received from continuously moving objects capable of reporting their position and velocity. Mobile applications—e.g., those that provide location-enabled services to mobile users—issue queries on this data.

3.1 Spatio-Temporal Data and Queries

The objects, represented as 2-dimensional points, update their positions periodically. As the server is recording the positions of a large amount of objects, updates should occur as rarely as possible. The current and anticipated future positions of the objects can be queried at any time. Therefore, continuous function that approximates the actual object movements and enables predictive queries is derived from the position data received.

An appropriate approximation function should satisfy the following requirements: (1) the parameters of the function can be obtained from the moving object; (2) the function reduces the amount of updates; (3) predicted positions are helpful in answering predictive queries; and (4) the function is easy to compute and its representation is compact.

It is common to predict an object's near-future position using a linear function of time [3, 4, 7, 8]. An object's position at time t is denoted by a 2-dimensional vector $\overrightarrow{P}$, and its velocity is given by a 2-dimensional vector $\overrightarrow{V}$. The function takes time as an argument, and returns the object's position:

$$\overrightarrow{P}(t) = \overrightarrow{P}(t_{\mathrm{up}}) + \overrightarrow{V}(t_{\mathrm{up}})(t - t_{\mathrm{up}}) \ . \tag{1}$$

Here t_{up} is the time of the last update, at which the object's position was $\overrightarrow{P}(t_{\mathrm{up}})$; $\overrightarrow{V}(t_{\mathrm{up}})$ is the velocity at time t_{up}, and $\overrightarrow{P}(t)$ is the predicted position at time t.

This function may be represented as a tuple $(\overrightarrow{P}(t_{\mathrm{ref}}), \overrightarrow{V}(t_{\mathrm{up}}))$, where time t_{ref} is an agreed upon, global reference time at which the object's position is stored. When an update of an object arrives at time t_{up}, its position $P(t_{\mathrm{ref}})$ at time t_{ref} is calculated using (1).

The linear function satisfies the four requirements for the approximation function. Velocity and position values are easy to obtain—they are output by GPS receivers [1], and the velocity can also be estimated based on previous positions (first requirement). The function's value is calculated in a constant time, and the representation is compact (fourth requirement). Studies show that using this function for vehicle positions, the average number of updates is reduced by more than a factor of two for accuracy thresholds below 200 meters, in comparison to the standard approach where the current position is assumed to be given by the most recently reported position [17] (second requirement). Finally, linear movement prediction offers better approximations of near-future positions than does stationary position prediction, yielding more reasonable answers to predictive queries (third requirement).

Three types of queries that a spatio-temporal index should support can be distinguished [3]. Let t, t_1, and t_2 be time points and let q_r, q_{r_1}, and q_{r_2} be 2-dimensional rectangles.

Q1 Timeslice query $Q1 = (q_\mathrm{r}, t)$ returns the objects that intersect with q_r at time t.

Q2 Window query $Q2 = (q_\mathrm{r}, t_1, t_2)$ returns objects that intersect with q_r at some time during time interval $[t_1, t_2]$. This query generalizes the timeslice query.

Q3 Moving window query $Q3 = (q_{\mathrm{r}_1}, q_{\mathrm{r}_2}, t_1, t_2)$ returns the objects that intersect, at some time during $[t_1, t_2]$, with the trapezoid obtained by connecting rectangles q_{r_1} and q_{r_2} at times t_1 and t_2, respectively. This query generalizes the window query.

Figure 1 offers an example encompassing four objects and three queries in 1-dimensional space. The *arrows* in the figure represent object movement. The queries $q1$, $q2$, and $q3$ are timeslice, window, and moving window

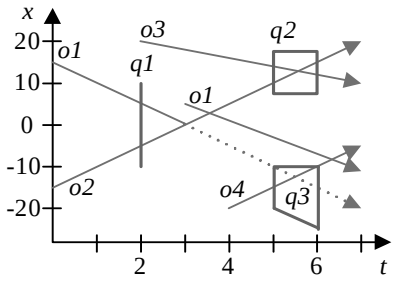

Figure 1: Example of objects and queries in a 1-dimensional space

queries, respectively. Query $q3$ has spatial ranges $q3_{r_1} = [-20, -10]$, $q3_{r_2} = [-25, -10]$, and time range $[5, 6]$. The result of the query depends on when the query is issued. If issued before time $t = 3$, the result is $\{o1\}$. If issued between time $t = 3$ and time $t = 4$, the result is the empty set $\emptyset$, because object $o4$ has not yet been inserted. Otherwise, the result is $\{o4\}$. Object $o1$ is updated at time 3 and its predicted trajectory changes. Its new trajectory does not intersect with the query.

3.2 Update Policies

The inaccuracy of the moving object positions available at the server side stems from two sources. The positions measured by the moving objects (e.g., using GPS) are inaccurate, and the use of sampling introduces inaccuracy. Because the measurement inaccuracy is much smaller than the sampling inaccuracy in a typical setting, we assume that the measurements are accurate and focus on the inaccuracy due to sampling.

In particular, we assume an approach where, at any point in time, the actual position of an object deviates from the position assumed on the sever side, the predicted position, by no more than a chosen distance threshold thr. An update policy should be adopted that satisfies the accuracy guarantee with as few updates as possible.

The so-called *point-based* update policy requires an object to issue an update when the distance between the object's current and its most recently reported positions reaches the threshold value. With this policy, the server assumes that an object remains where it was when it most recently reported its position. Frequent updates result.

To reduce the cost of updates a *vector-based* policy may be adopted [17], where each moving object shares a linear prediction, as given by (1), of its position with the server. When the distance between an object's actual and

predicted positions exceeds the distance threshold thr, the object issues an update to the server. The point-based policy is the special case of the vector-based policy, where the linear prediction function is constant ($\overrightarrow{V} = \overrightarrow{0}$, where $\overrightarrow{0}$ is the zero vector).

The point-based update policy is shown in Figure 2 (a). Here, the position $\overrightarrow{P}(t_i)$ is updated at time t_i, and the actual position remains in the circle with center $\overrightarrow{P}(t_i)$ and radius thr for some time, yielding a predicted position of $\overrightarrow{P}(t_i)$. At time t_{i+1}, the difference between the actual and predicted positions reaches thr, and an update is issued.

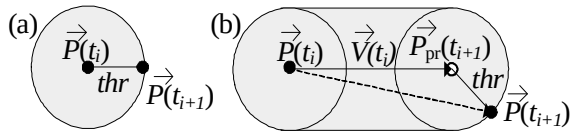

Figure 2: Point-based (a) and vector-based (b) update policies with accuracy threshold thr

Next, the vector-based policy is illustrated in Figure 2 (b). First, at time t_i, the object reports its actual position $\overrightarrow{P}(t_i)$ and velocity $\overrightarrow{V}(t_i)$ to the server. The server's prediction is illustrated by the *solid horizontal vector*. The object shares this prediction with the server. In addition, it repeatedly compares its actual position with the predicted position $\overrightarrow{P}_{\mathrm{pr}}$. When at time t_{i+1}, the object's position is $\overrightarrow{P}(t_{i+1})$, the distance between the two positions is thr, and an update is generated. Again updates are sent only when needed in order to maintain the accuracy guarantees.

As discussed in Section 3.1, the vector-based policy yields fewer updates than the point-based policy for the same accuracy guarantees and therefore is preferable.

3.3 Query and Data Enlargement

The notions of *precision* (p) and *recall* (r) [20] are commonly used for measuring the accuracy of a query result. The precision is the fraction of the objects in the result that actually satisfy the query predicate, and the recall is the fraction of the objects that satisfy the query predicate that are in the query result. Ideally, $p = r = 1$, meaning that the query result contains exactly the objects that satisfy the query.

However, the data are inaccurate—the positions of the objects are only known with accuracy thr. It is thus not possible to achieve $p = r = 1$; however, perfect recall ($r = 1$) can be achieved[1] and is a desirable requirement for an index. Thus, the query result is guaranteed to contain all objects that may satisfy the query predicate.

To achieve perfect recall, it is necessary to take the inaccuracy of the predicted positions into account. This may be done by means of either data or query enlargement.

Query enlargement addresses position inaccuracy by expanding the query area by thr in all directions. If different objects have different thresholds, the maximum threshold must be used. Perfect recall is achieved as all the objects that are actually in the query area have predicted positions that are no further than thr away from their actual positions.

The "fattened" query rectangle may be obtained as the Minkowski sum [21] of the two sets. Each point p_{q} that belongs to the query rectangle q_{r} is added to each point p_{s} that belongs to the segment s of length thr:

$$q_{\mathrm{r}} \oplus s = \{p_{\mathrm{q}} + p_{\mathrm{s}} | p_{\mathrm{q}} \in q_{\mathrm{r}} \wedge p_{\mathrm{s}} \in s\}$$

Figure 3 (a) shows query enlargement in a 2-dimensional space.

Next, with data enlargement object positions are expanded into spatial regions with extent. In particular, an object's position becomes a circle with radius thr, instead of being a point. The center of the circle is the predicted position. The object's actual position is always inside the circle. If the circle intersects with the query area, the object must be included in the query result. Figure 3 (b) illustrates data enlargement. The shaded area denotes the movement of the object.

A spatio-temporal index should support either query or data enlargement. However, existing indexes tend to ignore position inaccuracy and simply assume that they know the exact position of each object, meaning that $thr = 0$. Such indexes must be adjusted to index positions with non-zero threshold values.

[1] We note that perfect recall for queries that concern future times is only possible when updates that occur between the time a query is issued and the future times specified in the query cannot affect the query result.

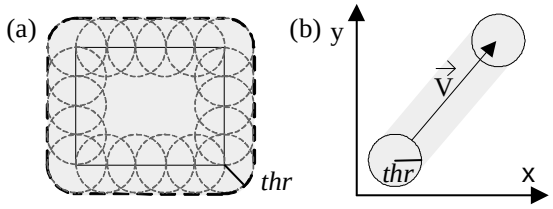

Figure 3: Example of query (a) and data (b) enlargement

4 Benchmark Data and Settings

The workload for an index consists of a sequence of the updates and queries. The benchmark specification contains definitions of workloads and procedures of using them. The desired properties of the workloads and workload generation are discussed first. Definitions of benchmark procedures, termed *experiments*, then follow.

4.1 Workload Parameters

A set of update and query parameters defines the benchmark workloads. The workloads aim to simulate a wide range of situations in which an index may be used. The following parameters are of interest:

Number of Objects The number of objects largely determines the size of the index and may be used to examine the scalability of the index.

Position and Velocity Skew These parameters determine the distribution in space of the object positions and velocities. They are highly related, as velocity skew leads to position skew. An example of skew is the concentration of stationary vehicles in the suburbs at night and in business districts during working hours, and many moving vehicles during the morning and afternoon rush hours.

Update Arrival Pattern The rate of updates depends on the chosen update policy as described in Section 3.2. With the vector-based policy, the durations in-between updates vary greatly. The update frequency depends on the movement trajectories and speeds of the objects. This parameter allows examination of how an index accommodates different frequencies of updates.

Position Accuracy Threshold The distance threshold thr (defined in Section 3.2) affects the update arrival rate and the query or data extents. By varying this parameter, the index ability to support various update frequencies as well as data and query sizes can be studied.

Query Parameters The required query types, their spatial and temporal extents and their time intervals are the query parameters of interest. The types of queries considered are described in Section 3.1.

Workload Duration The workload duration is measured as a number of updates executed by the index. This parameter allows examination of how an amount of updates affects the performance of an index.

4.2 Workload Generator

The workloads in the COST benchmark are generated using a workload generator that extends the generator developed by Šaltenis et al. [22]. That generator was chosen as the starting point because it is capable of easily creating workloads according to many of the parameters discussed in Section 4.1 and because it is fast in comparison to such generators as CitySimulator [23, 24] and GSTD [25, 26], which use complex functions, e.g., functions that control the interactions among the objects. We proceed to explain the original generator, then describe the extensions implemented.

A workload intermixes queries and updates with a chosen proportion. An index is then subjected to these operations. In the generator, object movement is either random or network-based. To accommodate the latter, a number of "hubs" with random positions and links between these form a complete, bi-directional, spatial graph. Objects move between hubs until the end of a simulation. The maximum speed of an object is chosen randomly from a set of maximum speeds. An object accelerates and decelerates when moving from one hub to another. Updates are generated in average intervals of $UpdateInterval$ time durations. For any kind of data, these parameters can be set:

$Objects$ Total number of moving objects.

$Space$ The extent of the space where the objects are moving.

$Speed_i, i = 1, ..., 50$ Set of maximum speeds of the objects. For each object, its maximum speed is chosen at random.

$TotalUpdates$ The number of update operations performed in the simulation.

$UpdateInterval$ The average duration between two successive updates of an object.

$Hubs$ The number of destinations between which the objects are moving. Value 0 implies uniform (random) distribution.

$QuerySize$ The maximum spatial extent of a query in percentages of the indexed space.

$QueryTypes$ The fractions of timeslice, window, and moving window queries (see Section 3.1). The sum of the three fractions must be equal to 1.

$QueryTime$ The maximum temporal extents of window and moving window queries.

$QueryWindow$ The maximum duration of time that queries may reach into the future.

$QueryingInterval$ Querying frequency relative to update operations.

$QueryQuantity$ The number of queries generated at each query generation event.

The generator was extended, enabling it to choose between its original update policy and the vector-based policy (as described in Section 3.2). The original policy was extended so that it is able to randomly select a different update interval for each object. Specifically, the generator was extended to accommodate three parameters:

Update Policy Specifies if the shared prediction based vector policy (0) or the original time-based (1) policy is used.

Threshold$_i, i = 1, ..., 50$ The threshold distance between the predicted and the actual positions, used in the vector policy. Up to 50 thresholds may coexist. For each object, its threshold is chosen at random. This parameter is used when *Update Policy*=0.

Update Interval$_i, i = 1, ..., 50$ The average duration between two successive updates of an object (as in the original generator). Up to 50 update intervals are possible. For each object, its average update interval is chosen at random. This parameter is used only when *Update Policy*=1.

With the vector-based update policy, updates are generated when the distance between the actual position of an object and the predicted position reaches *Threshold*$_i$. An additional update is generated when an object reaches a hub.

4.3 Evaluation Metrics

The COST benchmark uses two types of performance metrics: the average number of I/O operations per index operation, and the average CPU time per index operation (update, query). One I/O operation is one read of a page from disk or one write of a page to disk. Reads from and writes to the available main memory buffer are not counted as I/O operations. The CPU time for one operation is the time of CPU usage from the moment when the operation is issued to the moment when the result of the operation is computed. CPU measure is average time in milliseconds per operation. I/O is typically considered to be the main cost factor in determining an index's performance, while the CPU time is a minor factor.

5 Definitions of Experiments

A benchmark experiment is defined by a set of workload parameters and disk page and main memory buffer size settings. In each experiment, one parameter, or a set of related parameters, as defined in Section 4.1, is varied. The set of experiments was chosen with the objective of varying the important workload parameters from Section 4.1. Parameter values are chosen so that the workloads cover a wide variety of situations. To ensure that the benchmark stress-tests the indexes under study, some experiments use extreme parameter values. The page and buffer size settings are kept constant for all experiments.

The default values for all workload parameters and settings are listed in Table 1. The chosen values are commonly used in existing evaluations of spatio-temporal indexes (e.g., [4, 8]). The default speeds are typical speeds of vehicles, and the number of hubs simulates a real-world road network with a substantial number of destinations. The page and buffer sizes are relatively small, the objective being to obtain the effects of large indexes with relatively small volumes of data. For each experiment, described shortly, only parameters with values that differ from the defaults are listed. Note that it is possible to use only a subset of parameters $Speed_i$, $Threshold_i$, and $UpdateInterval_i$, $i = 1, ..., 50$, e.g., it is possible to assign the same speed to all objects by setting $Speed_1$ and omitting parameters $Speed_i, i = 2, ..., 50$.

All experiments measure the average CPU time and number of I/O's per operation.

Table 1: Default workload parameters and settings used in experiments

Parameter	Value	Parameter	Value
$Page$, $Buffer$	1 KB, 50 KB (50 pages)	$QueryingInterval$	400 updates
$Objects$	$100,000$	$QueryQuantity$	2 (in total 1000)
$Space$	$100,000 \times 100,000$ m^2	$QueryTime$	10 s
$Speed_i$, $i = 1, ..., 4$	12.5, 25, 37.5, 50 m/s	$QuerySize$	0.25% of $Space$
$TotalUpdates$	$200,000$	$QueryWindow$	50 s
$Hubs$	500	$QueryTypes$	0.6:0.2:0.2
$UpdatePolicy$	0	$Threshold_1$	100 m

Experiment 1. Number of Objects *Objective:* Examine index scalability.
Parameter values: $Points = 100, 200, ..., 1000$ K.
Number of workloads: 10.

Experiment 2. Position and Velocity Skew *Objective:* Examine the effects of position and velocity skew.
Parameter values: Part 1 (very high skew): $Hubs = 2, 4, ..., 20$. Part 2 (average skew): $Hubs = 20, 40, ..., 200$. Part 3 (low skew): $Hubs = 500, 1000, ..., 5000$, and 0 hubs (uniform distribution).
Number of workloads: 10 for parts 1 and 2, 11 for part 3.

Experiment 3. Maximum Speeds of Objects *Objective:* Examine the effects of varying maximum speeds as well as varying distributions of speeds among the objects. As fast objects are more likely to be updated than slow ones per given time unit, the update frequency increases with increasing speeds.
Parameter values: Part 1 (distribution of speeds): All objects are assigned either speed 25 m/s or 200 m/s, and workloads are generated so that the fractions of objects with speed 200 m/s are: 0.02; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 0.98. Thus, all $Speed_i$ are assigned either 25 m/s or 200 m/s, and for each workload, the smallest i is chosen that allows us to obtain the needed fraction of fast objects. Part 2 (low maximum speeds): $Speed_1 = 0.05; 2; 4; 6; 8; 10; 12; 14; 16; 18$. Part 3 (high maximum speeds): $Speed_1 = 30, 60, ..., 300$ m/s.
Number of workloads: 11 for part 1, 10 for the parts 2 and 3.

Experiment 4. Position Accuracy Threshold *Objective:* Examine the influence of varying thresholds as well as the distribution of varying thresholds among the objects. Note that the update rate depends on the threshold and that the simulation time increases as updates become infrequent.
Parameter values: Part 1 (distribution of thresholds): All objects are assigned either a threshold of 100 m or a 1000 m, and workloads are generated so that the fractions of objects with speed 1000 m are : 0.02; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 0.98. Thus all $Threshold_i$ are assigned either 100 m or 1000 m, and for each workload the minimum i is chosen that allows us to obtain the needed fraction of objects with large (and small) threshold.

Part 2 (equal thresholds for all objects): $Threshold_1 = 100, 200, ..., 1000$ m.
Number of workloads: 11 for part 1, 10 for part 2.

Experiment 5. Update Arrival Interval *Objective:* Examine the influence of varying update intervals as well as distribution of update intervals. The update frequency affects the time duration of a workload.
Parameter values: $UpdatePolicy = 1$. Part 1 (distribution of update intervals): Similarly to the two previous experiments, two values of a parameter, here $UpdateInterval_i$, are used—60 s (frequent) and 600 s (rare). The value of i is chosen so that workloads are obtained where the fractions of objects with an interval of 600 s are: 0.02; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 0.98. Part 2 (frequent updates): $UpdateInterval_1 = 20, 40, ..., 200$ s. Part 3 (rare updates): $UpdateInterval_1 = 120, 240, ..., 1200$ s.
Number of workloads: 11 for part 1, 10 for parts 2 and 3.

Experiment 6. Index Lifetime *Objective:* Examine the effect of varying index lifetime (in numbers of updates).
Parameter values: $TotalUpdates = 100, 200, ..., 1000$ K.
Number of workloads: 10.

Experiment 7. Query Types *Objective:* Examine the differences in performance for different types of queries: timeslice, window, and moving window queries.
Parameter values: $QueryTypes = 1:0:0, 0:1:0, 0:0:1$.
Number of workloads: 3.

Experiment 8. Query Parameters *Objective:* Examine the effects of varying spatial extents, temporal extents, and time windows of queries.
Parameter values: Part 1 (spatial extents): $QueryTypes = 0:1:0$, $QuerySize = 0.05, 0.15, ..., 0.95\%$. Part 2 (temporal extents): $QueryTypes = 0:1:0$, $QueryTime = 0, 20, ..., 120$ s. Part 3 (time windows): $QueryTypes = 1:0:0$, $QueryWindow = 0, 20, ..., 120$ s.
Number of workloads: 10 for part 1 and 7 for parts 2 and 3.

6 Application of the COST Benchmark

In order to ensure that the benchmark was well specified and yields useful results, it was applied for evaluating and comparing three existing indexes, namely the TPR-, TPR*-, and B^x-trees [3, 4, 8]. The TPR*- and B^x-trees were chosen because they are recent and represent the state of the art, and the TPR-tree is the predecessor of a dozen proposals for spatio-temporal indexes.

6.1 Introduction to the TPR-, TPR*-, and B^x-Trees

The TPR-tree (Time Parametrized R-tree) [3] and its descendant, the TPR*-tree [4], are based on the R*-tree [27]. These indexes are adapted for time-parametrized data and queries. Data objects are assigned to minimum bounding rectangles (MBRs) as in the R*-tree. Additionally, the TPR- and TPR*-trees use linear functions of time to represent the movements of the objects and MBRs.

Operations in the TPR-tree are handled similarly to the operations in the R*-tree, except that the penalty metrics of the R*-tree (e.g., MBR enlargement) are generalized to being integrals over a time period ranging from the current time and H time units into the future (calculated based on the update rate). The TPR-tree is optimized for timeslice queries.

The authors of the TPR*-tree have modified the TPR-tree by introducing new insertion and deletion algorithms. An additional heap structure is used during insertions with the objective of achieving better insertions. The *choose_path* algorithm selects an "optimal" (according to the paper's particular definition) path down the tree in insert operations. Instead of the integral used in the TPR-tree, the TPR*-tree calculates penalty metrics based on sweeping regions (the area covered by a moving MBR from the current time and H time units into the future). The TPR*-tree is optimized for moving-window queries.

The B^x-tree uses the B$^+$-tree structure and algorithms to store and retrieve data. Spatial data are transformed into 1-dimensional data using space-filling curves, e.g., the Hilbert or Z curves.

The B^x-tree partitions the time axis into intervals with a duration equal to the maximum duration in-between two updates of any object, Δt_{mu}. Each such interval is further partitioned into n phases. For each phase, an index partition is created. At any point of time, there exist at most $n + 1$ partitions. The partition in which to insert an

object is chosen according to the object's insertion time. As time passes, partitions expire, and new partitions are created. Objects in an expiring partition are reinserted into the newest partition.

Insertions, updates, and deletions are as in the B$^+$-tree. The index key of an object is calculated using the update time and the position of the object, which is stored as of the reference time of object's partition.

Queries in the B^x-tree must check each existing partition for qualifying objects. In one partition, the query area is first expanded by a factor of the current maximum $\vec{V}_{\max}$ and minimum $\vec{V}_{\min}$ projections and of the velocities $\vec{v}$ of all objects:

$$\begin{aligned} \vec{V}_{\max} &= (\max_{\vec{v} \in V}\{v^x\}, \max_{\vec{v} \in V}\{v^y\}), \\ \vec{V}_{\min} &= (\min_{\vec{v} \in V}\{v^x\}, \min_{\vec{v} \in V}\{v^y\}), \end{aligned}$$

where v^x and v^y are the projections of velocities $\vec{v}$ on x and y axes, and V is the set of velocities of current objects. Next, the expanded query rectangle may be reduced if the maximum and minimum velocities of the objects that fall into the expanded query area are smaller than $\vec{V}_{\max}$ or larger than $\vec{V}_{\min}$. These velocities are stored for each cell and each partition in a *velocity histogram* that is maintained in main memory.

For the experimental evaluation, the TPR- and TPR*-trees were extended to support data enlargement, and the B^x-tree was extended to support query enlargement. Enlarged data and query objects are approximated to squares and rectangles, respectively.

6.2 Conceptual Analysis of the TPR-, TPR*-, and B^x-Trees

With the aim of explicitly formulating expectations to the CPU and I/O performance of the indexes before experiments are conducted, this section presents conceptual analyses of the indexes. Section 6.3 experimentally evaluates the indexes.

In the ensuing discussion, we use "index performance" to refer to both CPU and I/O performance, unless explicitly stated otherwise. The workloads defined in Section 5 for the benchmark specification are considered. The analyses are based on the specifications of the indexes and the performance results provided in the articles that introduce to these indexes [3, 4, 8].

The papers that introduce to the indexes considered report on experimental evaluations and comparisons with one other index each. The authors of the TPR*-tree find that in most cases, the TPR*-tree performs better than the TPR-tree. The authors of the B^x-tree report that the B^x-tree outperforms the TPR-tree by a factor of 10 in many cases.

6.2.1 Index Size

The update and query performance of the TPR- and TPR*-trees are expected to degrade noticeably when the numbers of objects grow. As the space in which the objects are moving is limited, the density of the objects increases together with their numbers; thus, the overlap of MBRs is also expected to increase. Higher overlap presumably increases the I/Os and CPU time required for queries, as searches need to traverse more paths and inspect more nodes. For updates, the *choose_path* algorithm of the TPR*-tree might need to traverse more paths.

The B^x-tree is based on the B$^+$-tree, which degrades only slightly for both queries and updates when the amount of indexed objects grows. The B^x-tree is expected to experience only a slight update and query performance degradation with a growing amount of objects.

Hypotheses

H11 The update and query performance of the TPR- and TPR*-trees degrade noticeably when index size grows. The update performance of the TPR*-tree degrades more than for the TPR-tree, while the query performance of the TPR-tree degrades more compared to the TPR*-tree.

H12 The update and query performance of the B^x-tree degrades only slightly with increasing index size.

6.2.2 Position and Velocity Direction Skew

Increasing position and velocity skew are expected to positively effect the update and query performance of the TPR- and TPR*-trees. These trees index the data, not the underlying space, and they are balanced trees independently of the position distribution of the objects. Position and velocity skew are expected to concentrate MBRs in some areas, leading to smaller MBRs and less overlap among MBRs than for uniform data. An MBR expands more slowly if the movement directions of the objects in it are similar. With smaller MBRs that overlap less, fewer paths need to be traversed by queries and updates, yielding improved performance.

Position skew is only expected to slightly affect the update and query performance of the B^x-tree. Delete operations (also part of updates) may experience a problem if many objects in the same index partition fall into the same cell. Such objects are all assigned the same index key. This may result in many leaf nodes having to be read, until the required object is found. Query performance may be negatively affected by a high position skew, as all the objects that are in the cells that overlap with a query area have to be read from the disk. Many objects are then filtered out.

Direction skew should have no influence on the query performance of the B^x-tree, unless nearly all objects are moving in the same direction. A query region is expanded proportionally to the current maximum and minimum velocities, $\vec{V}_{\max}$, $\vec{V}_{\min}$, of the objects that fall into the expanded query area. If all the candidate objects move in the same direction, the query expansion is smaller in comparison with data with uniform velocity. However, if the objects are moving in several directions, skew should have no influence on the query performance.

Hypotheses

H21 The update and query performance of the TPR- and TPR*-trees improves as the position skewed of the data increases.

H22 The update and query performance of the B^x-tree is best for data with no or low position skew. If the skew is high, the update and query performance degrades. Extremely high skew may improve query performance.

6.2.3 Speed Skew and Maximum Speeds

The speeds at which the objects move are expected to have a high influence on the performance of the indexes.

The growth rates of the MBRs of the TPR- and TPR*-trees depend on the speeds of the objects in the MBRs. The presence of a fast-moving object results in the MBR of its leaf node and the MBRs that are in the path from the root to the leaf growing fast. However, no other MBRs are affected. The required I/Os and CPU time are expected to increase as the number of fast-moving objects increases, as more and more MBRs are affected.

The B^x-tree's performance depends on the global maximum and minimum velocities of the indexed objects. A query rectangle has to be expanded proportionally to the maximum and minimum velocities in all dimensions. If at least one very fast-moving object falls into the initially expanded query area, it is not possible to reduce the query in the object's movement direction. It is possible to reduce the query area only if the query covers cells with objects that move slower than the fastest objects among all the indexed objects (in all dimensions).

In general, all the indexes are expected to benefit from objects that move slowly.

Hypotheses

H31 The update and query performance of the TPR- and TPR*-trees and the query performance of the B^x-tree degrade when the maximum speeds of objects increase.

H32 The update and query performance of the TPR- and TPR*-trees degrade gradually when the numbers of fast-moving objects increase.

H33 The query performance of the B^x-tree degrades significantly for even a small number of objects with high velocities, compared to the situation when all the objects have low velocities. The update performance is not affected by the speeds of the objects.

6.2.4 Position Accuracy Threshold

The TPR- and TPR*-trees can use either query enlargement or data enlargement to support perfect recall, as described in Section 3.2. With data enlargement, it is possible to store different thresholds for different objects, and so data enlargement is expected be superior, and we consider only data enlargement. With data enlargement, increasing thresholds should degrade both update and query performance, as larger MBRs result.

The B^x-tree is capable only of query enlargement. When threshold values are different for different objects and/or parts of the space, the index must increase the query window by the maximum accuracy threshold in all directions. As a result, the B^x-tree should have worse query performance with larger thresholds than with smaller thresholds. The update performance should not be influenced.

Hypotheses

H41 All the indexes perform best when the upper bound of the threshold value is low. When query enlargement is used, only the query performance is affected. With data enlargement, both the update and query performance are affected.

H42 The B^x-tree's query performance degrades when there is at least one object that has a threshold value, compared to the situation where all threshold values are low.

H43 Due to data enlargement, the query performance of the TPR- and TPR*-trees is affected less by a small amounts of objects with high thresholds than is the query performance of the B^x-tree.

6.2.5 Update Arrival Pattern

The update arrival frequency affects the growth of MBRs in the TPR- and TPR*-trees. If updates are frequent, the MBRs are likely to be adjusted often, which improves the query performance. Because of its *choose_path* algorithm, the TPR*-tree is likely to adjust more MBRs than the TPR-tree during one update; thus, a lower number of objects that send updates frequently are needed to maintain tight MBRs. However, if updates are too frequent, access to the MBRs is likely to cost extra I/Os without significant benefits being obtained from the adjusting of the MBRs—the MBRs are tightened too often. If no object is updated in an MBR and not all of its objects are stationary, it grows infinitely large. Therefore, rare updates are expected to degrade the update and query performance of both the TPR- and TPR*-trees, especially the TPR-tree, as it tightens MBRs less frequently.

For the B^x-tree, objects that are updated rarely need to be migrated when an index partition expires. The B^x-tree reinserts objects that were not updated during the maximum update interval Δt_{mu}. If many objects need to be migrated, this will affect the average update performance, as the I/Os and the CPU time for migration are added as an update overhead in our evaluation. The query performance should not be affected by the update frequency. Only when the updates are extremely frequent and all objects reside in one index partition, the query performance should improve.

Hypotheses

H51 The update and query performance of the TPR- and TPR*-trees degrade when updates arrive very frequently or rarely. The best case lies somewhere in-between.

H52 The TPR-tree requires more updates to maintain tight MBRs in comparison to the TPR*-tree. The optimal update arrival frequency is higher for the TPR-tree than for the TPR*-tree.

H53 The update performance of the B^x-tree is better when updates arrive at the intervals equal to or shorter than the maximum update interval Δt_{mu} compared to longer intervals.

H54 The query performance of the B^x-tree improves when updates are much more frequent than the maximum update interval, compared to more rare updates. Otherwise, the query performance is not affected by the update arrival frequency.

6.2.6 Index Lifetime

The TPR- and TPR*-trees are expected to experience update and query performance degradation as the time of index usage passes because MBRs become less optimal with the passing of time. The TPR-tree may exhibit a higher performance degradation than the TPR*-tree, as the TPR*-tree tightens MBRs more often.

The B^+-tree's performance is nearly time-independent [8]. As the index lifetime increases, the B^x-tree is expected to experience only a very slight update and query performance degradation. The query performance of the B^x-tree is expected to be better at the very beginning of the index usage. This is so because when the index is first created, there exists only one index partition, and queries need to look for objects only in that partition. As time passes, more partitions are created that queries must access.

Hypotheses

H61 The update and query performance of the TPR- and TPR*-trees degrade as time passes.

H62 The update and query performance degrade faster for the TPR-tree than for the TPR*-tree as time passes.

H63 The update performance of the B^x-tree degrades very slightly as time passes. The query performance of the B^x-tree degrades faster at the very beginning of index usage and only very slightly later on.

6.2.7 Query Types

The TPR-tree is optimized for timeslice queries, while the TPR*-tree is optimized for window queries. In both indexes, queries are optimized for a time range H (time horizon, defined in Section 6.1). Window and moving window queries have larger extents (with respect to time) than timeslice queries, and window and moving window queries require more complex computation to determine whether the query area intersects with an MBR. Therefore, they are expected to require more I/Os and more CPU time in both trees.

The TPR*-tree is optimized for window queries. As a result, the difference between the performance of timeslice queries and the performance of window queries in the TPR*-tree is likely to be smaller compared to the TPR-tree. Still, timeslice queries should be executed with less I/Os, as they have no temporal extents.

The B^x-tree uses query expansion for all types of queries. The query performance depends on the query start and end times. Window queries are likely to be expanded more than timeslice queries, as they are expanded by the maximum expansion required for the query start and end times. Moving window queries are also expanded by the maximum expansion required for the query start and end times, but their area covers all regions in which the query rectangle moves. Timeslice queries are expected to require the least amount of I/Os and CPU time, window queries are expected to need slightly more, and moving window queries are expected to be the most expensive.

Hypotheses

H71 The query performance of the TPR- and TPR*-trees is best for timeslice queries, worse for window queries, and worst for moving window queries.

H72 The query performance of the TPR*-tree is affected less by the query types in comparison to the TPR-tree.

H73 The query performance of the B^x-tree is best for timeslice queries, worse for window queries, and worst for moving window queries.

6.2.8 Temporal Extents, Spatial Extents, and Time Windows of Queries

The TPR- and TPR*-trees are optimized for a time horizon H (see Section 6.1). The further into the future queries extend beyond H, the less optimal the query performance is expected to be due to expanding MBRs. A larger temporal extent is expected to require more I/Os and CPU time than a smaller extent.

In the B^x-tree, queries are expanded so that the fastest of all objects that might be in the query area is part of the query result. The B^x-tree can have at most one partition that has a reference time in the past. Other partitions have future reference times. That is, the B^x-tree is optimized for predictive queries. The query performance of the B^x-tree is expected to be better with queries that look further into the future than with queries that have a time close to the current time. An increasing temporal extent should degrade the index performance only slightly.

The B^x-tree partitions space into grid cells and uses a space-filling curve to enumerate the cells. Use of a high space granularity yields a small cell size and vice versa. The space-filling curve orders the cells. All the objects in the same index partition that fall into the same cell according to their positions are assigned the same index key. No matter how big the overlap between an expanded query rectangle and a cell is, all the objects that are in that cell have to be checked to determine whether they fall into the query area. For this reason, small queries (compared to the chosen space granularity) are expected to have a large overhead.

The TPR- and TPR*-trees do not need to choose any parameters in relation to the space granularity. The required I/Os and CPU time should increase gradually as the spatial extents of the queries increase.

Hypotheses

H81 The query performance of the TPR- and TPR*-trees are better for queries with a time close to the current time than for queries that look far into the future.

H82 The query performance of the TPR- and TPR*-trees are better for queries with shorter temporal extent than for queries with longer temporal extent.

H83 The performance of the B^x-tree is better for queries that look further into the future than for queries that have a time close to the current time.

H84 The query performance of the TPR- and TPR*-trees decrease proportionally to the query size.

H85 The performance of the B^x-tree has a large overhead for small queries and a smaller overhead for larger queries.

6.3 Experimental Evaluation Using the COST Benchmark

Implementations of the three indexes were obtained from their authors and modified where needed in order to perform the benchmark experiments. The indexes require a number of parameters to be set. For the B^x-tree, the maximum update interval is 120 s, there are 2 phases, and the cell size is 100×100 m^2. For the TPR and TPR*-trees, $H = 120$ s. All experiments were performed on a Sun Fire V880 server having 8x900 MHz CPU and 32 GB RAM running the Solaris 9 (SPARC) operating system.

6.3.1 Experiment 1—Index Size

Hypotheses H11 and H12 are examined in this experiment. The results are shown in Figure 4.

In comparison to the TPR*-tree, the TPR-tree exhibits a visibly better performance for updates, a visibly better CPU performance for queries, and a slightly better I/O performance for queries. The TPR*-tree was expected to perform better than the TPR-tree, as this was reported in the paper that presents the TPR*-tree [4]. The different results might be due to the different workloads that are used in the experiments. The density of objects in the experiments in [4] is 100 times higher than in our default setting.

As expected (Hypothesis H11), the update performance of the TPR*-tree degrades more than that of the TPR-tree. The TPR*-tree needs to traverse more branches than the TPR-tree during an insertion. When the overlap between MBRs increases, more branches might have to be traversed. However, contrary to expectation, the query performance of the TPR*-tree also degrades slightly more that is the case for the TPR-tree.

The B^x-tree, as expected (Hypothesis H12), degrades only slightly for queries and almost negligibly for updates when the index size grows, while the query performance of the TPR- and TPR*-trees (Hypothesis H11) degrade approximately linearly when the amount of objects grows. The reason is likely to be the greater overlaps among MBRs, resulting in more paths to be searched.

Hypothesis H11 is partially confirmed. The query performance of the TPR*-tree degrades slightly more than the query performance of the TPR-tree. Hypothesis H12 is confirmed.

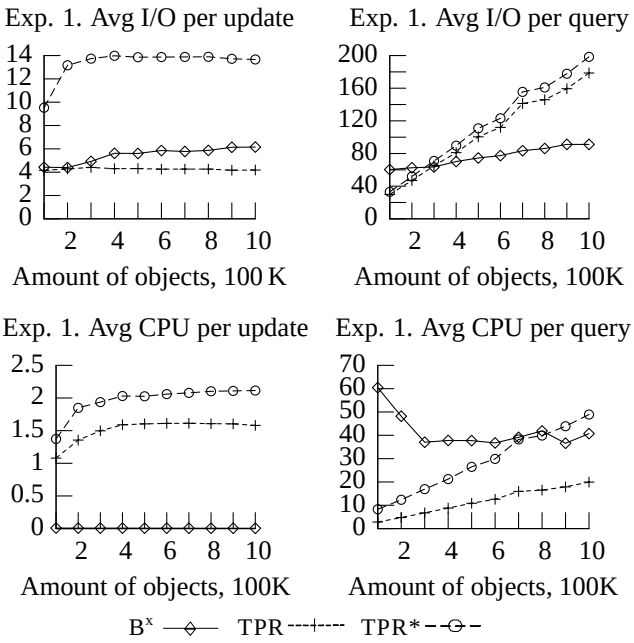

Figure 4: Experiment 1—index size

6.3.2 Experiment 2—Position and Velocity Skew

This experiment examines Hypotheses H21 and H22. The results are shown in Figure 5.

The query performance of the TPR- and TPR*-trees are significantly worse when position and velocity skew is very high compared to average and low skew. When the skew decreases from average to low, the query performance degrades slightly. Better performance was expected when the skew is high (Hypothesis H21). However, when the objects are concentrated in just a few places in the space, the overlap between the MBRs is likely to be very high, which leads to many paths having to be traversed in queries and updates. For settings with high position and velocity skew, updates become rare; thus, MBRs are adjusted less frequently and thus grow bigger.

As expected in Hypothesis H22, the query performance of the B^x-tree is slightly better with low skew than with high skew. Extremely high skew (2 hubs) does not influence the query performance negatively. This may be due to the similar movement directions of the objects. When the position and velocity distribution of the data approach uniform, the query performance improves significantly. A reason for this might be he shorter time that passes during the index lifetime. The effect of the index lifetime is observed and discussed in Experiments 4.2, 5.2, and 6.

The update performance of the B^x-tree degrades when skew is extremely high. This is due to rare updates, as objects move straight almost all the time which has the effect that the predicted velocities are similar to the actual velocities.

When position and velocity skew is average or low, the query performance of the B^x-tree is worse than the query performance of the TPR- and TPR*-trees. The average amount of I/Os per query is approximately double for the B^x-tree compared to the TPR- and TPR*-trees. This differs from earlier results [8].

Hypothesis H21 is not confirmed. The query and update performance of the TPR- and TPR*-trees are the best with average position and velocity skew, slightly worse with low skew, and significantly worse with very high skew. Hypothesis H22 is confirmed.

6.3.3 Experiment 3—Speeds of Objects

Hypotheses H31, H32, and H33 are examined in this experiment. The results are shown in Figure 6.

The query performance of the TPR- and TPR*-trees remain almost stable when the proportion of fast- and slow-moving objects changes. With only 2% fast-moving objects, the query performance is slightly better. When the number of fast-moving objects increases, the query performance slightly degrades in the beginning and then

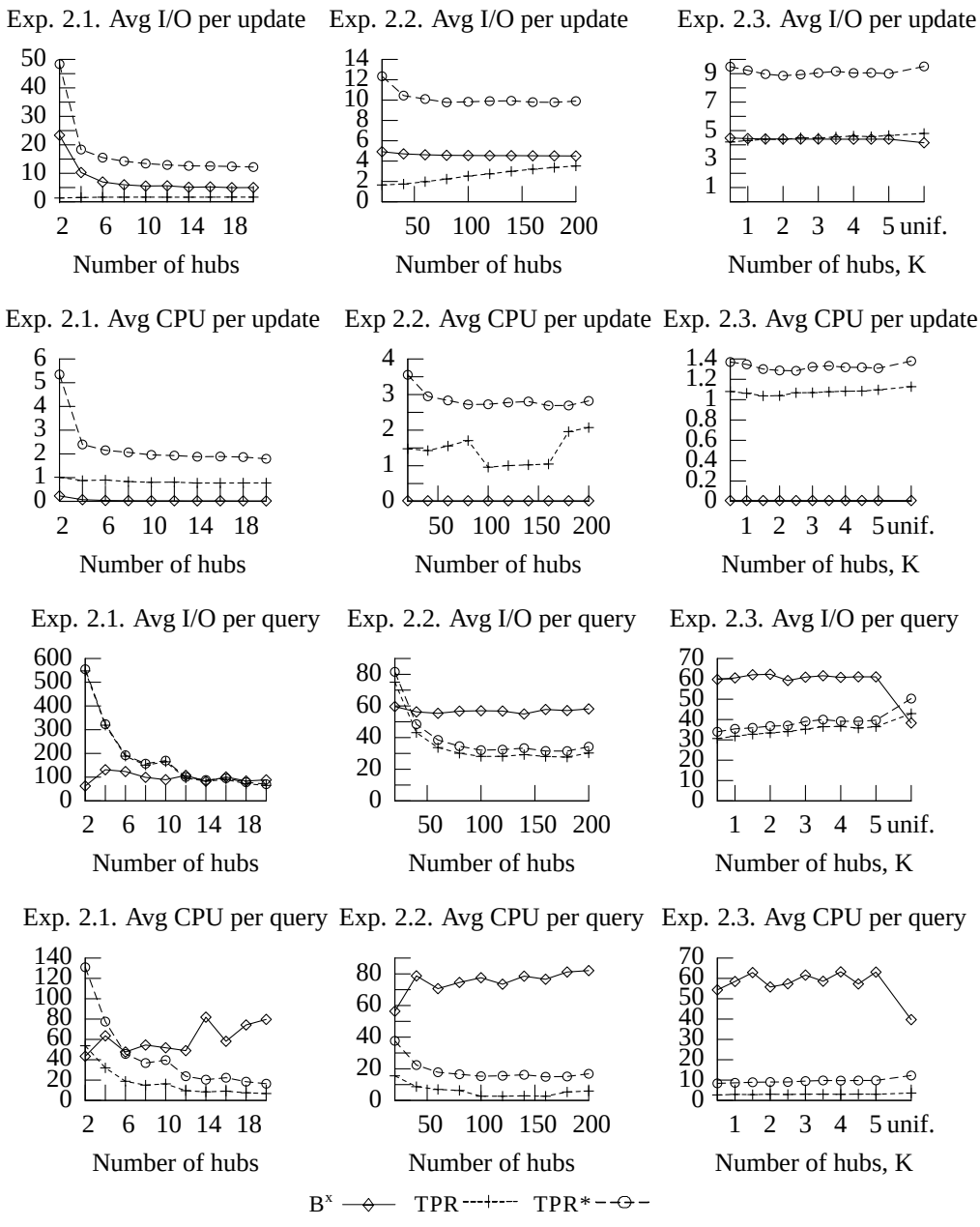

Figure 5: Experiment 2—position and velocity skew

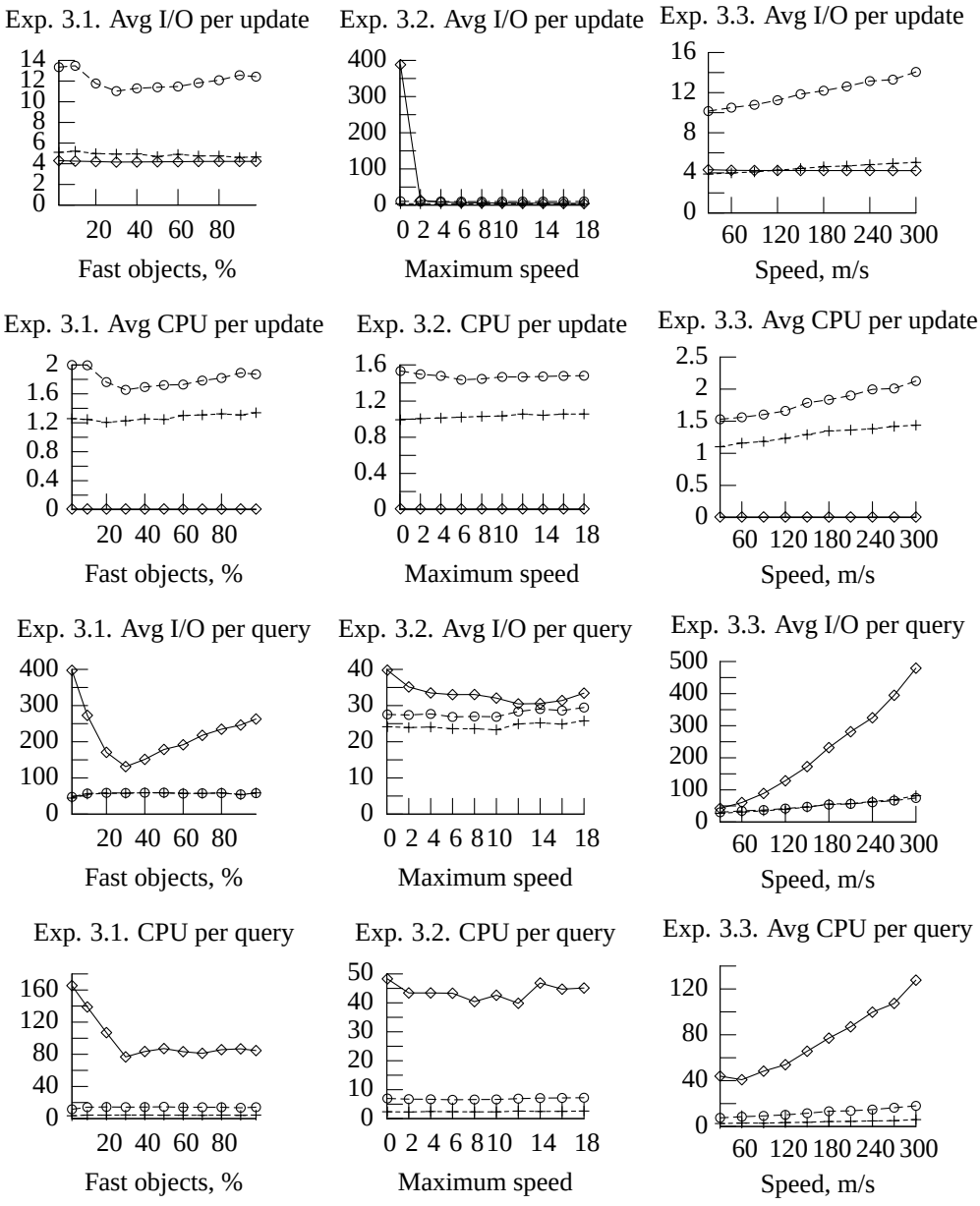

Figure 6: Experiment 3—speeds of objects

remains stable. Objects are assigned speeds independently on their position. One fast-moving object in an MBR is enough for the MBR to expand rapidly. Therefore, even a few fast-moving objects is enough to make most MBRs expand rapidly, which degrades the index performance.

Even with few fast-moving objects, the query performance of the B^x-tree degrades significantly. The combination of a longer index lifetime (due to low speeds and infrequent updates) and high maximum speeds yield a significant degradation of the query performance. When the index lifetime decreases (due to more fast-moving objects that are updated often), the query performance improves. However, when the amount of fast-moving objects increases further (above 30% of all objects), the performance again degrades due to large query expansions.

The query and update performance of the TPR- and TPR*-trees degrades only slightly when the maximum speeds increase, while the query performance of the B^x-tree degrades significantly. With the maximum speed of 300 m/s, the query I/O and CPU performance of the B^x-tree is more than 5 times worse than for the TPR- and TPR*-trees. This is due to large query expansions caused by fast-moving objects.

When speeds are very high, updates are very frequent, as the updates are generated according to the shared-prediction based update policy (Section 3.2). This helps to keep the MBRs tight; thus, the query and update performance of the TPR- and TPR*-trees degrades only slightly when the speeds of objects grow.

When the maximum speeds are extremely low (0.05 m/s), the update I/O performance of the B^x-tree is almost 100 times worse than the usual update performance of the B^x-tree. This is due to rare updates and the resulting high migration rate. The query performance does not degrade. With low speeds, query expansion is very small.

Hypothesis H32 does not hold. The performance of the TPR- and TPR*-trees are affected evenly by the presence of a small amount of fast-moving objects. Hypotheses H31 and H33 are confirmed.

6.3.4 Experiment 4—Accuracy Threshold

This experiment examines Hypotheses H41, H42, and H43. The results are shown in Figure 7.

The query performance of the B^x-tree is only affected slightly by a small fraction of large thresholds and is affected significantly more by the presence of many objects with large thresholds (above 60% of all objects) (Experiment 4.1). When the maximum threshold increases (Experiment 4.2), the query performance degrades gradually. Rare updates affect both query and update performance, due to migration and an increased index lifetime. This is discussed in the results of Experiments 4.2, 5.2, and 6.

The query performance of the TPR- and TPR*-trees degrade as well due to less frequent updates and thus larger MBRs. The update performance of the TPR- and TPR*-trees degrade when thresholds increase, a result of the MBRs being tightened less frequently. The update performance of the B^x-tree degrades when updates are rare, due to migration.

The query performance of all the indexes degrades when threshold values grow. However, the B^x-tree is more sensitive to increasing thresholds than the TPR- and TPR*-trees. When the threshold is 1 km, the average amount of I/Os per query is about 4 times higher for the B^x-tree than for the TPR- and TPR*-trees.

Hypothesis H43 does not hold. When the number of objects with high thresholds increases, the query performance of the TPR- and TPR*-trees degrades gradually. The query performance of the B^x-tree degrades significantly when there are many objects with large thresholds, but only slightly when there are few objects with large thresholds.

Hypothesis H41 is not confirmed. It states that the update performance is not affected when query enlargement is used. This is not true for the B^x-tree. In the B^x-tree, the update and query performance are affected due to more rare updates.

Hypothesis H42 is confirmed in part. The B^x-tree performs worse when there is at least one object with a high threshold. However, when the amount of objects with high thresholds increases, the query and update performance degrade as well.

6.3.5 Experiment 5—Update Arrival Interval

We next examine Hypotheses H51, H52, H53, and H54. The results are shown in Figure 8.

The query and update performance of all the indexes degrades when the update frequency decreases. The results differ from the expectations. Even very short update intervals (20 s) have a positive influence on the query and update performance of the TPR- and TPR*-trees. Frequent access of MBRs is compensated by tight MBRs.

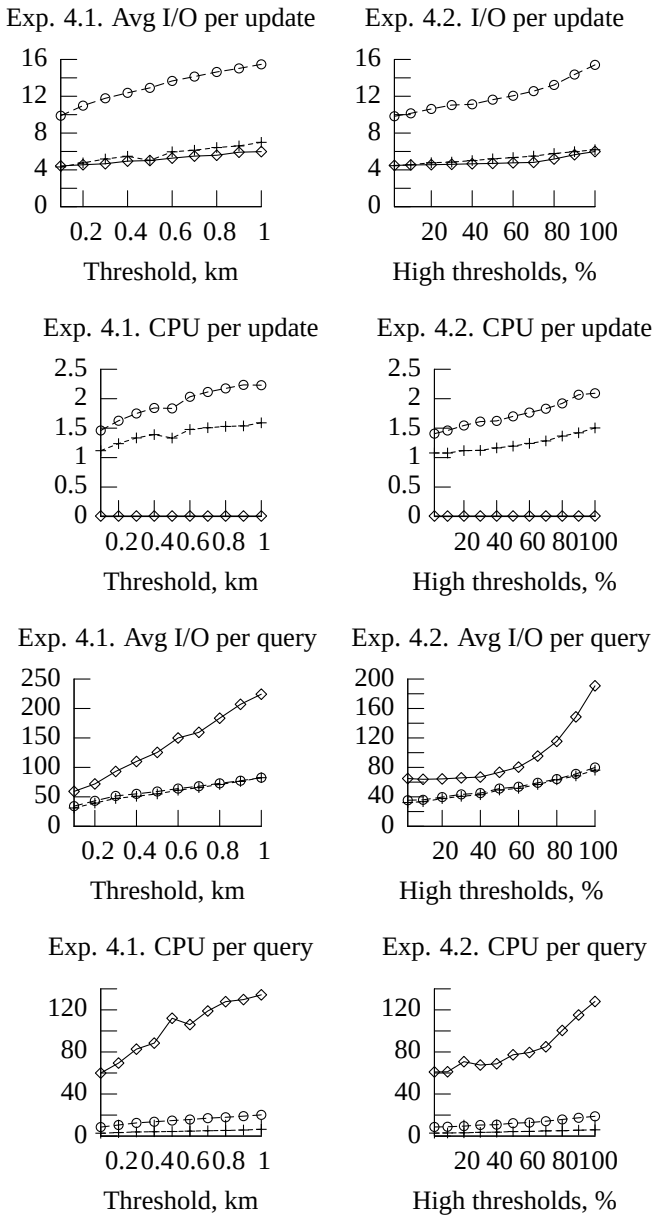

Figure 7: Experiment 4—accuracy threshold

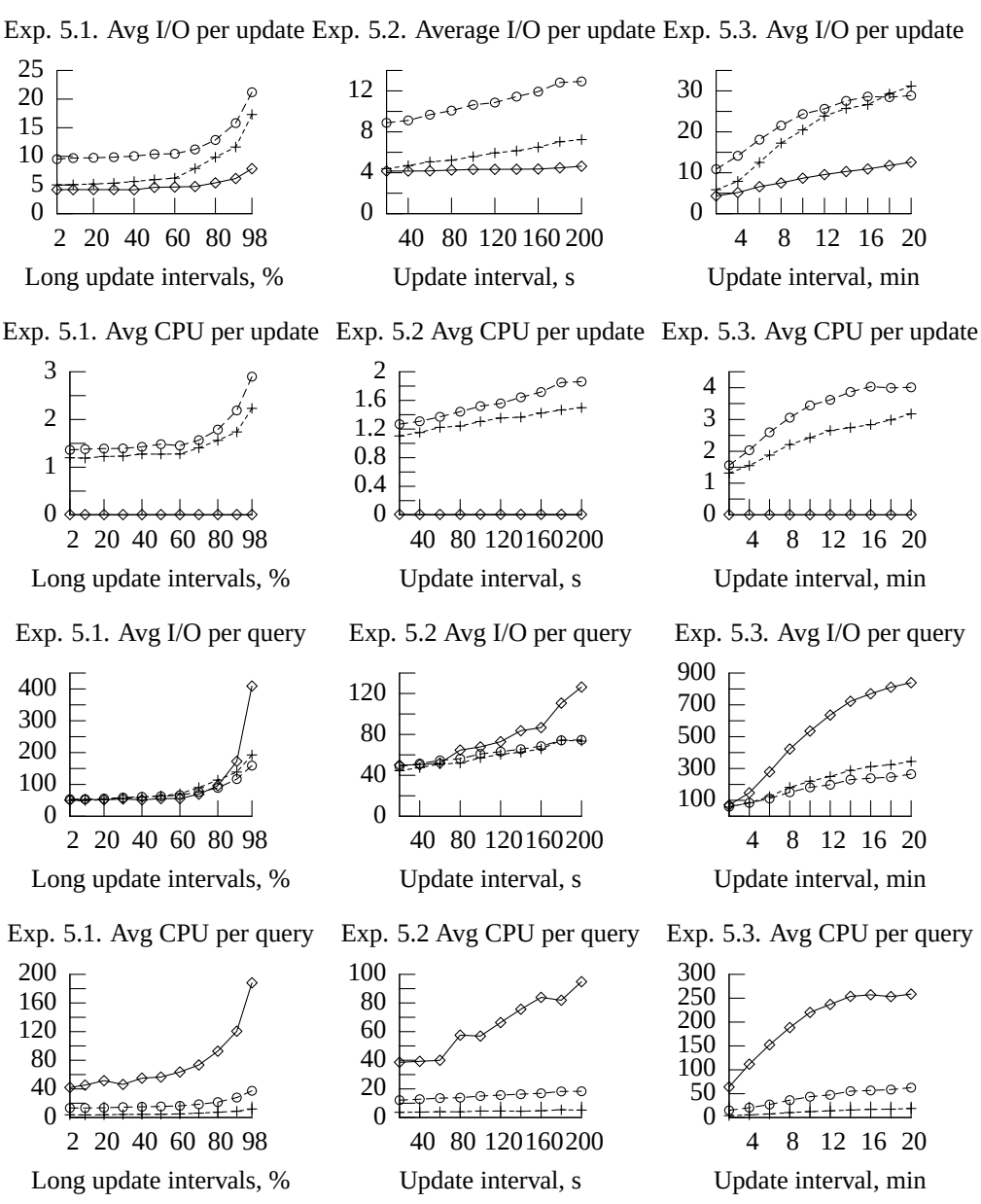

Figure 8: Experiment 5—update arrival interval

The query performance of the B^x-tree degrades significantly when the update interval increases. This contrasts the expectation that the query performance remains stable (Hypothesis H53). It is interesting to notice that there usually is a jump in the query and update performance when the average update interval length reaches the timestamp of a new index partition. This is true when the update interval is rather short (Experiment 5.2). In the generated data, each object receives 2 updates on average. With an update interval length of 20 s, at most two index partitions are used. As a result, the queries need to traverse only two partitions. When the update interval length grows, the third partition comes into use. After the first partition expires, migration is introduced. The objects that are migrated to a new partition update the histogram of velocities (defined in Section 6.1). This increases the query expansion in the new partition.

The update performance of the B^x-tree degrades as the update interval increases, due to migration.

Hypotheses H51, H52, H53, and H54 do not hold. The query and update performance of all the indexes degrades when the update frequency decreases. The query performance of the B^x-tree degrades significantly, while the query performance of the other indexes degrade only slightly. The update performance of the TPR- and TPR*-trees degrade more than that of the B^x-tree.

6.3.6 Experiment 6—Index Lifetime

Next, we examine Hypotheses H61, H62, and H63. The results are shown in Figure 9.

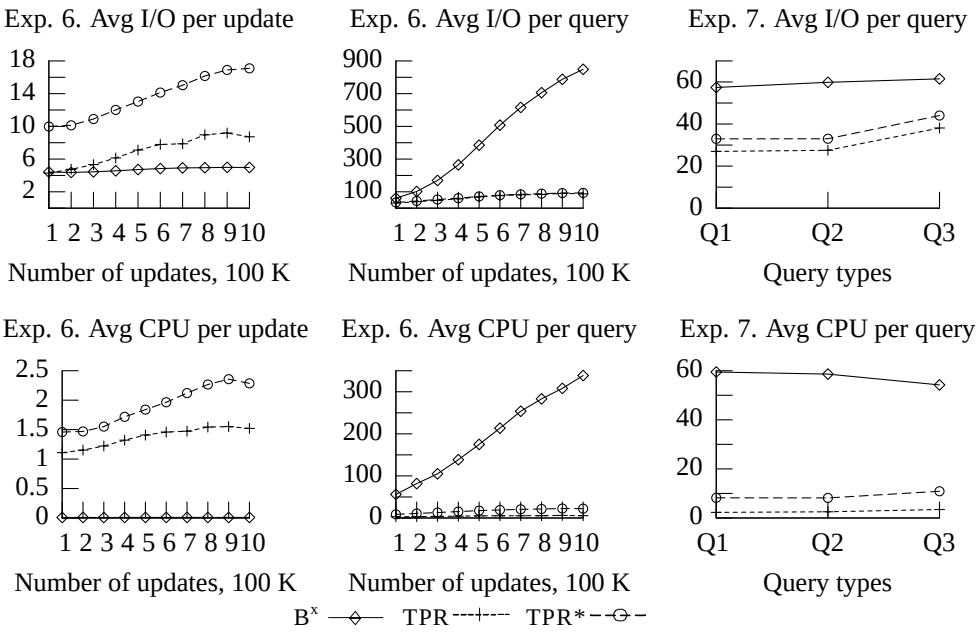

Figure 9: Experiment 6—index lifetime; and Experiment 7—query types

The query performance of the B^x-tree degrades significantly as the index lifetime increases, while the update performance is not affected. The query performance degradation is also observed and discussed in Experiments 4.2 and 5.2.

The update and query performance of the TPR- and TPR*-trees degrade slightly as the index lifetime increases. The degradation is the most visible for the TPR*-tree. As time passes, the MBRs in both trees expand. The update operations in the TPR*-tree become more costly because more paths need to be traversed.

With 100 K updates, the index lifetime is approximately 140 s. As $\Delta t_{\mathrm{mu}} = 120$, migration is needed only once. With 1,000 K updates, the index lifetime is about 40 min. With a small amount of updates, the objects do not have enough time to accelerate (see the generator description in Section 4.2). After some time, their speeds are likely to be higher and lead to a bigger query expansion. The reasons for the low performance when the index lifetime increases are discussed further in Experiment 5.2.

Hypothesis H61 is confirmed. Hypothesis H62 does not hold. The update and query performance of the TPR- and TPR*-trees degrade at the same rate as time passes. Hypothesis H63 also does not hold. The query performance of the B^x-tree degrades significantly as time passes, while the update performance remains stable.

6.3.7 Experiment 7—Query Types

We now consider Hypotheses H71, H72, and H73. The results are shown in Figure 9.

The query performance of all the indexes are almost independent of the query type. However, the slight differences for the different types that are seen are as expected.

Hypotheses H71 and H73 are confirmed. Hypothesis H72 does not hold. The performance of the TPR- and TPR*-trees change equally when the query types change.

The influences of the time windows and temporal extents of queries on the query performance are discussed in Experiments 8.2 and 8.3.

6.3.8 Experiment 8—Spatial Extents, Temporal Extents, and Time Windows of Queries

This section examines the last hypotheses: Hypotheses H81, H82, H83, H84, and H85. The results are shown in Figure 10.

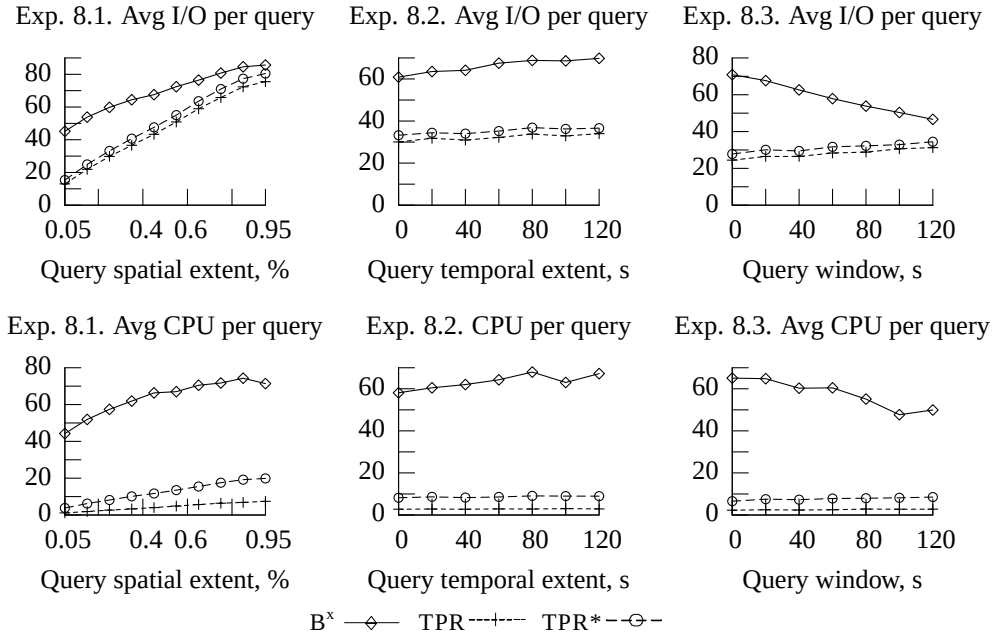

Figure 10: Experiment 8—spatial extents, temporal extents, and time windows of queries

The I/O and CPU performance of the TPR- and TPR*-trees increases approximately proportionally to an average spatial extent of queries. This means that small queries do not have a noticeable overhead. For small queries, the TPR- and TPR*-trees perform significantly better than the B^x-tree.

The B^x-tree is less sensitive to changes in spatial extents of queries than are the TPR- and TPR*-trees. When the initial query area is small, the query enlargement of the B^x-tree due to thresholds and velocities introduce large overheads. With large spatial extents (above 0.8% of the space), the average number of I/Os per query is almost equal for all the indexes. However, the CPU time of the B^x-tree is relatively higher.

The query performance of the indexes are almost independent of the temporal extents of the queries. This was expected for the B^x-tree. The TPR- and TPR*-trees were expected to perform better for queries with shorter temporal extents (Hypothesis H82). This can be explained as follows. Timeslice and window queries select about the same amount of objects. The average spatial extents of queries (5×5 km) is much larger than the distance an object can travel during the maximum temporal extents of queries (250 m).

The TPR- and TPR*-trees, as expected (Hypothesis H81), perform better when the time windows of queries are small. When the times are closer to the current time, the MBRs for such queries are smaller than for queries that look far into the future. However, the difference in performance is very slight.

As expected (Hypothesis H83), the performance of the B^x-tree improves when the time windows of queries increase and become close to the reference timestamp of the last phase. It is expected that objects update their positions within the maximum update interval Δt_{mu}, and the majority of the objects are indexed in the partitions with the latest future timestamps. Therefore, the queries that look further into the future have to be expanded less than the queries with timestamps close to the current time.

Hypotheses H81, H83, H84, and H85 are confirmed. Hypothesis H82 is confirmed partially. The query performance of the TPR-, TPR*-, and B^x-trees degrade when the temporal extents of queries increase, but the degradation is hardly noticeable.

6.4 Summary of the Experimental Evaluation

The experiments demonstrate that the benchmark fulfills its purpose: it is capable of uncovering strengths and weaknesses of the indexes (only some of which are reported by the papers that introduce the indexes). For example, the experimental results identify situations in which the B^x-tree has lower query performance than the TPR-tree. As another example, the benchmark shows that situations exist where the TPR-tree outperforms the TPR*-tree for updates.

The B^x-tree exhibits a substantial query performance degradation when the maximum speeds of objects increase, when the intervals in-between the updates grow to be long, when the position accuracy threshold becomes very large, or when the index lifetime is long. The index performs well for both queries and updates when the index lifetime is very short and speeds are low or average. The main reasons for the query performance degradation are likely to be the unadjusted maximum update interval and the large query expansion. The reasons for the large expansions are the high maximum and the low minimum velocities recorded in the histogram of velocities, and a high threshold.

The TPR- and TPR*-trees exhibit very similar query performance in most cases, which is somewhat contrary to earlier experimental results [4]. The most likely reason for this mismatch is the different types of workloads used. The queries used in the earlier experimental evaluation of the TPR*-tree are not only moving, but also expanding. In addition, the underlying space 100 times smaller than in the present study. This is likely to result in many overlapping MBRs. High maximum speeds of objects result in overlapping MBRs as well. Figure 6 shows the results of Experiment 3.3, where the maximum speeds of objects are high. In this experiment, the TPR*-tree performs slightly better than the TPR-tree.

The B^x-tree seems to be a good choice when the number of objects is big, the maximum interval in-between the updates is known, the accuracy threshold is low, and the speeds of the objects do not exceed the usual speeds of vehicles.

In other cases, the TPR- or TPR*-trees should be chosen. The choice between the TPR- and TPR*-trees should be made by taking into account the expected query workload and the density of objects: the TPR-tree performs better with timeslice queries and low object densities, while the TPR*-tree performs better with expanding queries and high object densities (according also to experimental results reported elsewhere [4]).

The TPR- and TPR*-trees appear to be the most versatile indexes; however, the B^x-tree is based on the B^+-tree, which is already available in many DBMSs. Therefore, the creation of a more robust version of the B^x-tree may be a promising research direction.

7 Conclusions and Future Work

A number of indexes for the current and near-future positions of moving objects exist, and more are underway. This state of affairs creates an increasing need for a neutral and well-articulated experimental setting for evaluating and comparing these indexes.

This report proposes a benchmark, termed COST, that is targeted specifically toward the evaluation of such indexes. The benchmark aims to make realistic assumptions about the experimental settings—data is inherently inaccurate, predictive queries that reach into the future are covered, the indexes are assumed to be stored persistently on disk. More specifically, an update technique is assumed where positions are guaranteed to be accurate

within agreed-upon thresholds and where updates occur only when necessary in order to satisfy the guarantees. The indexes may use either query or data enlargement to account for the inaccurate data. The benchmark includes a workload generator, definitions of experiments, and evaluation metrics. It considers a wide range of workload parameters that cover many real-world situations.

As proof of concept and to evaluate the benchmark, it was applied to the TPR-, TPR*-, and B^x-trees. The experiments demonstrate that the benchmark is well-specified and is capable of covering a wide range of situations. Weaknesses and strengths of the indexes were detected by examining the sensitivity of the indexes to workloads with varying parameter values, including workloads with extreme settings. The experimental results cover situations that were not covered in the papers that introduced the indexes, due to more extensive experiments. The obtained results provide guidance as to when each of the indexes should and should not be used.

The benchmark may be extended by inclusion of such aspects as index size in disk pages, I/Os and CPU time for bulkloading and bulk operations, and evaluation of concurrent accesses.

Further analysis of the support for uncertainty in the indexes is also warranted. In this paper, variation among the thresholds of the objects was considered. As an extension of this, thresholds may be varied across time and space. The development of update, as well as query and data enlargement policies, for such workloads is an interesting research direction.

Further studies of existing spatio-temporal indexes are also warranted, possibly including detailed studies of special cases and aspects specific to individual indexes. Examples include detailed studies of overlaps among MBRs, growth rates of MBRs, and the grouping of objects into MBRs in R-tree-based indexes. For the B^x-tree, such studies may cover query enlargement aspects and migration loads. For all indexes, it is of interest to investigate aspects such as tree depths and node fanouts. Studies such as these have the potential to offer insights that may guide the development of improved indexes.

Acknowledgments

This research was conducted within the project Telematics Applications Based on Ubiquitous Sensor Networks, funded by the Electronics and Telecommunications Research Institute, South Korea. C. S. Jensen is also an adjunct professor in Department of Technology, Agder University College, Norway.

References

[1] Blewitt, G.: Basics of the GPS technique: observation equations. Geodetic Applications of GPS (1997) 10–54

[2] Wikipedia: GPRS (2001–2005) `http://en.wikipedia.org/wiki/GPRS`.

[3] Šaltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the positions of continuously moving objects. In: Proceedings of the 21st ACM SIGMOD International Conference on Management of Data. (2000) 331–342

[4] Tao, Y., Papadias, D., Sun, J.: The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In: Proceedings of the 30th International Conference on Very Large Data Bases. (2003) 790–801

[5] Procopiuc, C.M., Agarwal, P.K., Har-Peled, S.: STAR-tree: an efficient self-adjusting index for moving objects. In: Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments. (2002) 178–193

[6] Šaltenis, S., Jensen, C.S.: Indexing of Moving Objects for Location-Based Services. In: Proceedings of the 18th International Conference on Data Engineering. (2002) 463–472

[7] Patel, J.M., Arbor, A., Chen, Y., Chakka, V.P.: STRIPES: an efficient index for predicted trajectories. In: Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data. (2004) 635–646

[8] Jensen, C.S., Lin, D., Ooi, B.C.: Query and update efficient B+-tree based indexing of moving objects. In: Proceedings of the 30th International Conference on Very Large Data Bases. (2004) 768–779

[9] Zobel, J., Moffat, A., Ramamohanarao, K.: Guidelines for presentation and comparison of indexing techniques. SIGMOD Rec. **25** (1996) 10–15

[10] Gray, J., ed.: The Benchmark Handbook for Database and Transaction Processing Systems. Morgan Kaufmann Publishers, Inc. (1993)

[11] Theodoridis, Y.: Ten benchmark database queries for location-based services. The Computer Journal **46** (2003) 713–725

[12] Myllymaki, J., Kaufman, J.: DynaMark: A Benchmark for Dynamic Spatial Indexing. In: Proceedings of the 4th International Conference on Mobile Data Management. (2003) 92–105

[13] Werstein, P.F.: A performance benchmark for spatiotemporal databases. In: Proceedings of the 10th Annual Colloquium of the Spatial Information Research Centre. (1998) 365–373

[14] Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y.: Benchmarking access methods for time-evolving regional data. Data Knowl. Eng. **49** (2004) 243–286

[15] Cheng, R., Kalashnikov, D.V., Prabhakar, S.: Querying imprecise data in moving object environments. IEEE Trans. on Knowl. and Data Eng. **16** (2004) 1112–1127

[16] Tao, Y., Cheng, R., Xiao, X., Ngai, W.K., Kao, B., Prabhakar, S.: Indexing multi-dimensional uncertain data with arbitrary probability density functions. In: Proceedings of the 31st International Conference on Very Large Data Bases. (2005) 922–933

[17] Čivilis, A., Jensen, C.S., J. Nenortaitė, J., Pakalnis, S.: Efficient tracking of moving objects with precision guarantees. In: Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services. (2004) 164–173

[18] Wolfson, O., Sistla, A.P., Chamberlain, S., Yesha, Y.: Updating and querying databases that track mobile units. Distrib. Parallel Databases **7** (1999) 257–387

[19] Pfoser, D., Jensen, C.S.: Capturing the uncertainty of moving-object representations. In: Proceedings of the 6th International Symposium on Spatial Databases. (1999) 111–132

[20] Lazaridis, I., Mehrotra, S.: Approximate selection queries over imprecise data. In: Proceedings of the 20th International Conference on Data Engineering. (2004) 140–152

[21] Weisstein, E.W.: Minkowski sum. From MathWorld—A Wolfram web resource (1999–2005) `http://mathworld.wolfram.com/MinkowskiSum.html`.

[22] Šaltenis, S., Jensen, C.S., Leutenegger, S., Lopez, M.: Indexing the positions of continuously moving objects. Technical report, Aalborg University (November 1999)

[23] Kaufman, J., Myllymaki, J., Jackson, J.: CitySimulator (2001) `https://secure.alphaworks.ibm.com/aw.nsf/techs/citysimulator`.

[24] Myllymaki, J., Kaufman, J.: LOCUS: A testbed for dynamic spatial indexing. IEEE Data Eng. Bull. (Special Issue on Indexing of Moving Objects). **25** (2002) 48–55

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

[26] Theodoridis, Y., Silva, J.R.O., Nascimento, M.A.: On the generation of spatiotemporal datasets. In: Proceedings of the 6th International Symposium on Advances in Spatial Databases. (1999) 147–164

[27] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data. (1990) 322–331

[28] Jensen, S., Tiešytė, D., Tradišauskas, N.: Spatio-temporal workload generator (2004) `http://www.cs.aau.dk/~dalia/generator.htm`.