Open Access

Open Journal of Databases (OJDB)
Volume 3, Issue 1, 2016
http://www.ronpub.com/ojdb
ISSN 2199-3459

High-Dimensional Spatio-Temporal Indexing

Mathias Menninghaus [A], Martin Breunig [B], Elke Pulvermüller [A]

[A] Institute of Computer Science, University of Osnabrück, Albrechtstraße 28, 49069 Osnabrück, Germany, {mathias.menninghaus, elke.pulvermueller}@uos.de

[B] Geodetic Institute, Karlsruhe Institute of Technology, Englestr. 7, Karlsruhe, Germany, marting.breunig@kit.edu

Abstract

There exist numerous indexing methods which handle either spatio-temporal or high-dimensional data well. However, those indexing methods which handle spatio-temporal data well have certain drawbacks when confronted with high-dimensional data. As the most efficient spatio-temporal indexing methods are based on the R-tree and its variants, they face the well known problems in high-dimensional space. Furthermore, most high-dimensional indexing methods try to reduce the number of dimensions in the data being indexed and compress the information given by all dimensions into few dimensions but are not able to store now - relative data. One of the most efficient high-dimensional indexing methods, the Pyramid Technique, is able to handle high-dimensional point-data only. Nonetheless, we take this technique and extend it such that it is able to handle spatio-temporal data as well. We introduce a technique for querying in this structure with spatio-temporal queries. We compare our technique, the Spatio-Temporal Pyramid Adapter (STPA), to the R^{ST}-tree for in-memory and on-disk applications. We show that for high dimensions, the extra query-cost for reducing the dimensionality in the Pyramid Technique is clearly exceeded by the rising query-cost in the R^{ST}-tree. Concluding, we address the main drawbacks and advantages of our technique.

Type of Paper and Keywords

Regular research paper: *high-dimensional, spatio-temporal, databases, indexing, access methods, subway-track planning*

1 Introduction

When modeling real-world objects it is often necessary to not only include three spatial but also a temporal dimension as our world is three-dimensional and time is always moving forward. Objects change over time and these changes may be modeled along a temporal axis. A rather simple way to describe changes over time is the usage of a valid-time model [38]. For example, when modeling building structures, railway tracks or similar architectures, the valid-time of a modeled object describes when the object exists and when not. If the valid-time-span of a building object lies after the current time of the modeled reality, then the building is planned, i.e. it does not yet exist. If the valid-time-span of a building object lies before the current time of the modeled reality it existed once, but has been destructed already. Hence, if the valid-time-span of a building object contains the current time it is valid and therefore exists at that very moment. For such a valid-time model changes on an object in the real world mean, that the valid-time of the modeled object ends with the time of change and a new object is created, whose valid-time begins with the time of the change. A chain of changes is therefore described as a chain of objects whose valid-time-spans are touched by the valid-time-spans of the antecedent and subsequent

versions of the object. When modeling the exchange of a houses' windows, not the entire house would be destroyed and rebuilt in the model, but the parts which changed, thus the windows.

When modeling objects with temporal dimensions as described above, one must bear in mind that time is modeled different to space. All temporal dimensions refer to an increasing *now*-value. By including time one must also include the continuous progression of time in the real world. Additionally, objects may not only be marked with constant time values such as dates and times, but also as *current*. Thus in contrast to the spatial part of an object which always has a certain beginning and end in every dimension, the temporal part may not have a certain end. For example, if the destruction of a building is not predetermined, the end of its valid-time should be set to *current*. That means that it might be destroyed one day but not *now*. Usually, *current* is a flag and every time the end of a time dimension has been marked as *current*, the end time is set to the current value of *now* during computation. The different attempts to model such a *current* flag is addressed in the next section.

Despite three spatial and the above described temporal dimension, models of real world applications may include additional dimensions. Databases may store different versions of the inserted objects, i.e. the time of insertion, update and deletion. Similarly to the valid-time, this *transaction*-time is connected to an increasing *now*-value. The transaction-time-span of a newly inserted object starts with the current time and ends with a flag like *until changed*, which denotes that the object has not yet been deleted. When the object is updated or deleted, the end of its transaction-time-span is set to the current time. Note that the current time is not the above used flag *current*, but the value of *now* at the moment of insertion, update, or deletion. The connection of valid- and transaction-time to the bi-temporal [38] model is discussed in the following section.

Additional to these five useful dimensions for real-world models, three spatial and two temporal, a model may contain a lot of additional thematic attributes and the spatial extent of a real world object may be described in different ways. For example, the aforementioned buildings may be represented by a building process model, a discrete model, or both. Also the more or less detailed spatial parts of an object in different levels of detail may be stored with one and the same object. This leads to at least three additional spatial dimensions for every level of detail [11].

In order to keep the redundancy of the different spatial representations, levels of detail, and versions of an object at a minimum, an indexing method needs to provide direct access to all informations instead of spreading the informations on several specialized indexing methods. Additionally, the indexing method should avoid costly joins between query results on several sub-indices whereat one may store spatial, one temporal, and one thematic data. Moreover, when using separated indices, it is difficult to tell how to split the information up into several indices, since some queries focus on the spatial difference between different versions of an object, others on thematic changes or the impact on different levels of detail etc.

Storing objects of such a model with a great amount of dimensions including several temporal dimensions into a database requires an indexing method which is not only able to handle high-dimensional but also spatio-temporal data efficiently. Especially, the indexing method must be able to store and process representations of the aforementioned *now*-relative data. In this paper, we propose a new indexing method which is called *Spatio-Temporal Pyramid Adapter (STPA)* and is able to process these values and handle the combination of both spatio-temporal and high-dimensional data well. Despite building models and construction plans, the desired indexing method may be used for any application which deals with spatio-temporal data with a discretely changing spatial extend and a high number of dimensions. For instance, classic GIS applications to manage cadastre data where additional dimensions may be pricing, land use and different levels of detail may use this method. CAD applications which are used to generate the aforementioned building models and construction plans may also benefit from a faster access. Even image processing applications, which are used to compare historic pictures with current images taken from the same position and angle may benefit from a spatio-temporal, high-dimensional indexing method.

We give an overview of spatio-temporal and high-dimensional indexing methods in section 2. We also address the main drawbacks of the existing approaches when confronted with high-dimensional spatio-temporal data. In order to classify different access methods several attempts for both, spatio-temporal and high-dimensional data, have been made which include specific datasets and benchmarking applications. We give a brief overview over these evaluation techniques in section 2.3. Following the experiences from section 2, we describe the design of our new indexing approach in section 3.1. A concrete implementation is described in section 3.3. The evaluation in section 4 includes the description of the concrete implementations of the R^{ST}-tree and our evaluation program together with the evaluation of the STPA in comparison to the R^{ST}-tree. We conclude our work in section 5 giving an outlook to future enhancements and additional areas of applications.

2 RELATED WORK

In this paper, we deal with high-dimensional data as well as with spatio-temporal data. We give an introduction about temporal data in the previous section. This is specified in this section along with a literature overview about spatio-temporal and high-dimensional indexing methods and their evaluation.

2.1 Spatio-Temporal Access Methods

One may specify at least two temporal dimensions: on the valid-time axis it is specified whether or not a modeled object exists in reality, i.e. is valid. The transaction-time describes the appearance of an object in a database system. Thus, the transaction-time of an object describes when the object has been inserted and when it has been deleted or updated. A model supporting both aspects is considered to be bi-temporal. These temporal aspects were described in detail by Snodgrass [38]. Worboys [46] propose a model for spatial and temporal information and, amongst others, motivated it with the building process of road networks which is quite similar to the planning of subway tracks, the application the STPA will mostly be used for [9].

All applications which are using temporal aspects have to deal with some special cases which arise when modeling the continuous progression of time. They need to be able to represent the ongoing present, which is often denoted by a flag called *now* or similar. The semantics of *now* in databases are discussed in detail by Clifford et al. [13]. Especially, if one does not only need to store certain timestamps, but also values growing along with the ongoing *now*, the flag *current* may be used. That means that a field, which is set to the value *current*, will somehow change over time. Thus the relation between an object whose temporal value has been the current time or *now* at insertion time to an object whose temporal value is *current* will also change. An optimistic approach sets a value to a very large number or even infinity instead of *current*. In terms of the end of the valid-time of an object this imposes the assumption that the object is always valid. Problems arise when the end value of the valid-time-span changes to a certain time. In this case, all previous queries regarding the future validity of the object must be considered to be wrong. Among the definition of this and other problems, Clifford et al.[13] propose a framework for working with bi-temporal data.

Numerous approaches exist which grant fast access to such data which take the above mentioned problems into account. Most of these approaches are designed to handle continuously moving objects. However, we do not deal with moving objects like vehicles or even hang-slopes, where objects change their position in space and not their general appearance. Instead we deal with non-moving yet discretely changing data. For instance, consider building plans which change over time or buildings which are changed during the building process and altered after they have been built. Abraham and Roddick [2], Mokbel et al. [26], Nguyen-Dinh et al. [29] and Pelekis et al. [31] give good overviews about past and recent approaches on indexing methods for continuously moving objects.

Spatio-temporal indexing methods have been developed mostly as extension of well known access structures such as the Quadtree by Finkel and Bentley [17] which has been extended by the Overlapping Linear Quadtree from Tzouramanis et al. [41]. Or the B-tree by Bayer and McCreight [4] which has been extended to the B^X-tree by Jensen et al. [22]. Most of the spatial indexing methods depend on the R- [19] and R^*-tree [5], which is widely used in spatial information and database systems and seem to be the best choice for handling massive low-dimensional data. For instance, van Oosterom et al. [43] proofed the performance of the R-tree on several database systems.

The improvements on the R-tree regarding spatio-temporal data can be separated into two branches. First, there are indexing methods that combine several R-trees in order to take the *now*-relative temporal intervals into account. The Historical R-tree by Nascimento and Silva [27] is an R-tree of R-trees, one for every time step, where new R-trees only store changed objects and use references to the subtrees of the unchanged nodes in the previous R-tree in order to save space. The 2+3 R-tree by Nascimento et al. [28] uses one 2-dimensional R-tree to store the current spatial information and one 3-dimensional R-tree to store all past data, i.e. every object that already has *now*-relative temporal intervals. If every state of the object is known a priori, the 2+3 R-tree is reduced to a 3-dimensional R-tree. Both are only able to handle one time dimension.

Second, some indices try to enhance the R-tree with spatio-temporal functionality by changing its insert, split and delete algorithms. The R^{ST}-tree by Saltenis and Jensen [34] uses time-parametrized values and an additional splitting algorithm to take growing time-spans for the transaction- as well as the valid-time into account. One extension of the often-cited Time-Parametrized-R-tree (TPR-tree) [36], the R^{EXP}-tree [35], takes moving objects into account by simply using integrals for the R^*-tree operations such as *union*, *overlap*, *volume*, and *margin* which denotes the length of the boundaries of a rectangle. With this it is able to build up conservative minimum bounding rectangles (MBR) which take the future extension of the spatio-temporal MBRs into account. Therefore, it is not necessary to update the MBRs

with every change of the object's shape and position. The R^{EXP}-tree does not use a constant parametrization value but a dynamical computation via a time horizon function. The time horizon function can be used to enhance the R^{ST}-tree which is shown in section 4. As there seems to be no other indexing method which is designed for discretely changing spatial and bi-temporal data, the R^{ST}-tree is the most suitable competitor for the STPA-technique. Stantic et al. [39] propose a new indexing technique for temporal data based on the relationships between intervals [3], the TD-tree. Despite the fact that this technique does not incorporate *now*-relative data, it is used by He et al. [20] to create a parallel indexing technique for spatio-temporal data. Although it does not support *now*-relative data, it provides another perspective on indexing high-dimensional spatial data which is partly used by the STPA.

2.2 High-Dimensional Access Methods

In the past, the R-tree [19] seemed to be the best choice not only for indexing spatial but also point and spatio-temporal data. Nonetheless, the performance of the R-tree and its best known enhancement, the R^*-tree [5], decreases rapidly when used for objects with a higher number of dimensions (> 10). Berchtold et al. [8] state, that the overlap of the directory nodes of a R^*-tree increases rapidly for uniformly distributed points with increasing dimensionality. They propose the X-tree, which extends the R^*-tree with a new splitting technique and supernodes. If a node cannot be split such that the splitted nodes have a minimum overlap, the capacity of the original node is enhanced instead.

The maximum number of entries in one node of the R-tree or a similar hierarchical structure decreases with an increasing number of dimensions. Thus, more nodes are needed to index high-dimensional data and therefore more nodes and blocks are accessed when querying the structure. Referring to that, the TV-tree [25] reduces the number of dimensions by using a telescoping function. The number of dimensions used in the directory nodes to discriminate the path to the leafs is reduced significantly.

In contrast to these R-tree-Based methods, many methods use regular space partition in order to divide the multi-dimensional data space into subspaces. The Quadtree [17] method for instance recursively divides the n-dimensional data space into 2^n sub-spaces. The Grid-File [30] uses a directory file, which contains pointers on the buckets that partition the data space. For indexing point data, the K-D-B-tree [32] partitions the data space into point pages and merges them into region pages. These region pages are recursively merged up to a root page which represents the whole data space. Space-partitioning methods do not face the problem of overlapping regions but the number of partitions grows exponentially [45] with an increasing number of dimensions and therefore also have a decreasing performance. Facing these problems, Weber et al. [45] propose the VA (*vector-approximation*) file which divides the data space into 2^b rectangular cells, where b is a user defined number of bits. Together with a formular for approximatively addressing each data point and a filtering function for efficiently excluding data cells when querying, the VA-file outperforms the X- and R-tree and works even better in higher dimensions.

Beside R-tree-based and space-partition-based methods, dimension-reducing methods use to map the n-dimensional data points or rectangles onto a one dimensional value and store these with a B^+-tree [14] or similar method. iDistance [21] identifies the n-dimensional points by the nearest reference point and the distance to this reference point. The PL-tree [44] uses a scaling function to map a real vector to an integral vector and the bijective cantor pairing function to map these n-dimensional data points into a scalar. It outperforms the X- and R-tree but is outperformed by iDistance in terms of query performance.

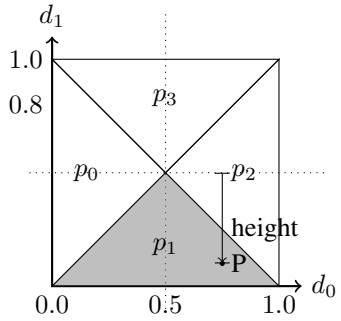


Figure 1: Calculation of the pyramid value of a 2-dimensional point $P(0.75, 0.1)$. The data space has been divided into 4 pyramids. The pyramid value for P is 1.4: 1 because it lies in pyramid 1 plus $.4$ because this is its distance (height) to the center (0.5) in dimension d_1. (after Berchtold et al. [7])

The Pyramid Technique [7] maps n-dimensional points into one-dimensional values. It therefore splits the n-dimensional data space into $2n$ n-dimensional pyramids, whose bases are the borders and whose centers are the center of the data space. The pyramid-value is calculated as follows (Figure 1): the places before the decimal point depict the pyramid in which the point lies and the decimal places are the height in that pyramid. The height of a point in pyramid p is the distance of that point from the center in dimension $p \; MOD \; n$. After mapping all points to their pyramid values they are stored into a B^+-tree [14], using the pyramid values as keys, while

the leaf nodes contain the original n-dimensional key. As the calculation of the pyramid value is not bijective, a mapped range query on data stored with the pyramid technique may result in more elements than expected. Therefore, one has to test every match against the original query. Moreover, an n-dimensional range query is converted into up to $2n$ one-dimensional queries because every pyramid which intersects the query rectangle has to be queried. The main drawback of the Pyramid Technique is, that one has to certainly know the borders of the data space and rebuild the complete index if data is stored which lies outside the originally assumed borders. This rebuild causes an overhead of node accesses. However, experiments indicate that this overhead becomes negligible for high-dimensional data.

The Pyramid Technique can be enhanced to the Extended Pyramid Technique [7] by shifting the center of the pyramids to the median of the data set in every dimension. Doing so, the efficiency of querying on clustered data sets is improved, but the center of the pyramids needs be altered if its distance to the real median is to high. Therefore, an approximation of the real median of the already inserted data is tracked by a histogram and constantly compared to the actual center. Altering the center to the median requires a complete rebuild of the structure which can be done most efficiently by using bulk loading techniques on the underlying B^+-tree. Zhang et al. [47] generalized this technique to the P^+-tree, which dynamically divides the data space into several subspaces in order to deal with more than one cluster of data points.

2.3 Evaluating High-Dimensional and Spatio-Temporal Indices

Together with high-dimensional and spatio-temporal access methods several datasets and benchmarking applications have been proposed in order to analyze and evaluate different indexing approaches. For the generation of spatio-temporal data, i.e. moving-objects data, the best known frameworks may be GSTD [40], OPORTO [33], and G-TERD [42]. Brinkhoff [10] propose a framework for generating network-based moving objects e.g. traffic in road networks, just like the data created by SUMO [6]. In addition, Jensen et al. [22] (COST), Düntgen et al. [15] (BerlinMOD), and Chen et al. [12] define benchmarks for moving objects indices. As there seems to be no data generator or benchmark for spatio-temporal data with discretely changing spatial extent, the experiences with the above mentioned generators and benchmarks are used to create a new workload generator in section 4.

The most recent approach for the analysis of indexing techniques for high-dimensional point data is QuEval [37, 23], a framework which can be extended with index structures, data sets, and distance metrics. Unfortunately, it is designed for high-dimensional point and not spatial or even spatio-temporal data.

3 High-Dimensional Spatio-Temporal Indexing

An indexing method is needed which is able to handle high-dimensional and spatio-temporal data. This method needs to incorporate *now*-relative data and eliminate the drawbacks of other spatio-temporal approaches which are a high overlap between directory nodes and the shrinking number of entries in each directory node with an increasing number of dimensions. In this section, we describe the design and implementation of a method which is capable of doing so.

3.1 Design

First, the overlap between directory nodes should be minimized. The simplest way to accomplish that, is to exclude the possibility of an overlap between different nodes by a space partitioning method. Second, the size of the directory nodes should be constant at an increasing number of dimensions. This is possible by mapping the intervals with start and end point in every dimension to a single value and storing all elements with that value as key into a B^+-tree.

None of the techniques described in section 2.2 is able to store *now*-relative data. Therefore, the mapping function needs to map the *current* flag as well. It also needs to map in respect to the ongoing time and the fact, that the time correlated to *current* is constantly growing.

Mapping *now*-relative data to a single constant value causes a cluster of data points at this value for large spatio-temporal data sets. For this reason, the indexing method needs to be able to convert the *current* flag to a constant value for a specific dimension, and then map the data with this converted value into a single value and not convert the whole data to a constant value. To our knowledge, the only method which already contains a constant value which can be used as a replacement for the *current* flag in every dimension, is the Pyramid Technique. As it uses the Pyramid Technique as indexing method in its core, the new indexing method described in this paper is called Spatio-Temporal Pyramid Adapter (STPA). Before mapping a point to the corresponding pyramid value, the STPA converts every *current* value to the corresponding median value which is the center of the pyramids in the given dimension. Using the center of the pyramids as mapping for the *current* flag has the additional benefit, that the Pyramid Technique works best if the queries contain the center of the pyramids

```
1  class STPA {
2
3    BPlustree bplustree
4    double-array medians
5
6    method insert (rectangle){
7      double pyramidValue = convert(rectangle)
8      bplustree.insert(pyramidValue, rectangle)
9      // update histogram, rebuild index if medians change
10   }
11
12   // methods delete, update, and lookup work analogously
13
14   method convert (rectangle) {
15     instantiate double-array point with a length of 2*rectangle.dimensions
16     for i = 0, i < rectangle.dimensions, i++ do
17       point(dim) = rectangle.begin.get(i)
18       if(rectangle.end.get(i) == current)
19       then
20         point(i+rectangle.dimensions) = medians(i + rectangle.dimensions)
21       else
22         point(i+rectangle.dimensions) = rectangle.end.get(i)
23     end
24     double pyramidValue = convertToPyramidValue(point); // use [7]
25     return pyramidValue
26   }
27 }
```

Figure 2: Pseudo-code for inserting a new element with the STPA. First the rectangle is converted into the corresponding pyramid value and then inserted into an underlying B$^+$-tree using the pyramid value as key.

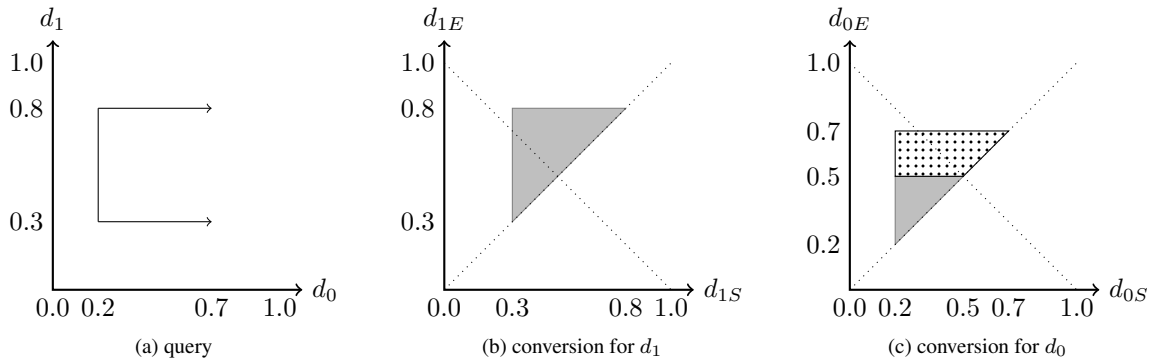
[47]. And, by definition, the Pyramid Technique is not only a dimension-reducing but also a space-partitioning method and therefore no overlaps between the directory nodes occur. The incorporation of the ongoing time is discussed in section 3.2. Note, that the Pyramid Technique is only suitable for point data. So at first the STPA needs to convert rectangular data into point data.

Hence, before inserting new data, given an n-dimensional data space and spatio-temporal data, the STPA converts the n intervals of one data key into an $2n$-dimensional point. The first n entries of that point are the start values of the intervals and the second n entries are the end values of the intervals. Afterwards, the *current* flags are replaced by the median value in the given dimension. Finally the $2n$-dimensional point is converted into the corresponding pyramid value and stored into an underlying B$^+$-tree, just like in the original Pyramid Technique. The pseudo-code for inserting new elements is shown in Figure 2.

Summing up, the STPA uses a B$^+$-tree, which contains key-value pairs, where the original n intervals of the MBR and a pointer to the original object are the value, and the corresponding pyramid value is the key. The additional space needed to store the data is $O(N)$ for the pyramid values of N inserted objects plus the additional space needed for the B$^+$-tree. Lookup, deletion, and update work just like insertion: First the data is converted into the corresponding pyramid value and then the operations are proceeded as for the original B$^+$-tree.

Converting the n-dimensional intervals into the pyramid value lies in $O(n)$, as the algorithm needs to determine the pyramid and the distance to the center by a constant number of visits of each dimension. Nonetheless, we do not detect any impact of the conversion process in our evaluation in section 4. The query conversion, which is described in section 3.2 also lies in $O(n)$, but the query itself always is a range query on a B$^+$-tree with no additional costs. As for the conversion to pyramid values, we do not detect any measurable impact of our query conversion on the performance.

However, shifting the center of the pyramids in order to minimize the distance to the real median of the already



(a) query (b) conversion for d_1 (c) conversion for d_0

Figure 3: Example query (a) in two dimensional space with one temporal (d_0) and one spatial (d_1) dimension and its conversion to two during queries (b+c). The current value of *now* is 0.7. The median, e.g. the center of the pyramids, in each dimension of the converted space is 0.5. After conversion the space of the query according to d_0 is extended because the query rectangle ends with *current* in that dimension.

inserted data causes a complete rebuild of the structure, because every element has to be converted again. The costs of this operation may be reduced by using bulk-loading techniques and setting the initial center to a good approximation of the expected median value. Berchtold et al. [7] state that a rebuild of the structure is so unlikely that it does not have an impact on the overall performance. Therefore, we do not investigate the impact of a rebuild any further.

3.2 Querying

Querying elements with the STPA is more difficult than insertion or deletion because of two reasons. First, an n-dimensional query needs to be converted into a $2n$-dimensional range query because the original Pyramid Technique is only capable of that type of query. Second, queries containing *now*-relative values or queries which might match *now*-relative data need to be converted respectively. Hereinafter, we describe the query process up to the conversion into $4n$ range queries. This range query is converted as suggested for the original Pyramid Technique. For a better understanding of the following conversion steps consider the following example.

Example: Given a two-dimensional normalized space, with one spatial dimension d_1 and one temporal dimension d_0, get all elements which are contained in the query rectangle $((0.2, current), (0.3, 0.8))$ which is shown in Figure 3a.

Because the data is stored as intervals, the STPA first needs to convert the query into interval queries. Allen [3] and Kriegel et al. [24] suggested 13 relationships between intervals which are visualized in Figure 4.

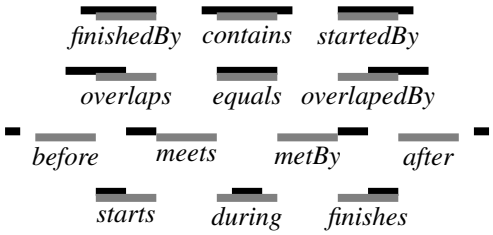


Figure 4: Thirteen general interval relationships (after [3, 24]). The gray line is the query interval, the black line a matching interval to that query.

He et al. [20] combine these relationships to the 8 fundamental relationships between two n-dimensional objects suggested by Egenhofer [16]. Likewise, the STPA determines which interval queries are affected by the given n-dimensional query and union the ranges of these interval queries. In contrast to the approach given by He et al. [20] it is not efficient to split the query into several sub queries and combine the results with logical operators like *AND* or *OR*. The STPA therefore either needs to join the possibly large result sets of the sub queries or track which elements already have been matched by a query. He et al. [20] suggest to use a flag for every element and every dimension to depict if an element has already been visited by a query in that dimension. This also means that this flag has to be reseted after every query. Just like the join of the result sets of every sub query in the first variant, this reset requires each queried element to be called again and causes a large overhead of I/O operations. Table 1 shows which interval query is affected by which of the fundamental relationships.

Table 1: List of the general interval relationships which are affected by the 8 relationships between n-dimensional objects (after [20]).

Relationships between objects [16]	Interval relationships [3]
Disjoint	Before, After
Meets	Meets, MetBy
Overlaps	Overlaps, OverlappedBy
Equals	Equals
Contains	Contains
Contained	During
Covers	Covers
CoveredBy	CoveredBy

Example: Following Table 1 the given *contained* query needs is converted into two *during* queries.

After deciding which interval queries have to be used, the STPA converts this interval queries into range queries. Figure 5 shows how a one-dimensional interval query (Q_S, Q_E), where Q_S is the start and Q_E is the end of the query interval, is converted into a two-dimensional range query. This gives us the $2n$ ranges to be queried by the underlying Pyramid Technique.

Example: The during queries are created as shown in Figure 3b+c. Before the special cases for *now*-relative data are incorporated all *current* values are set to the median value at that dimension. In the example the median in each dimension is 0.5.

Before querying the STPA also needs to incorporate *now*-relative data into the generated range queries for the following two cases:

1. The original n-dimensional interval query itself contains *now*-relative values, thus the end value in at least one dimension equals *current*.
2. The interval query matches the current value of *now*.

For every one-dimensional interval query (Q_S, Q_E) the STPA has to determine how to change the converted two-dimensional range query, if one of the aforementioned cases sets in. Table 2 lists how to adapt the resulting region query for each of the query types from Figure 5. Note that the *now*-relative values have been stored as the center value of the pyramids in the specific dimension.

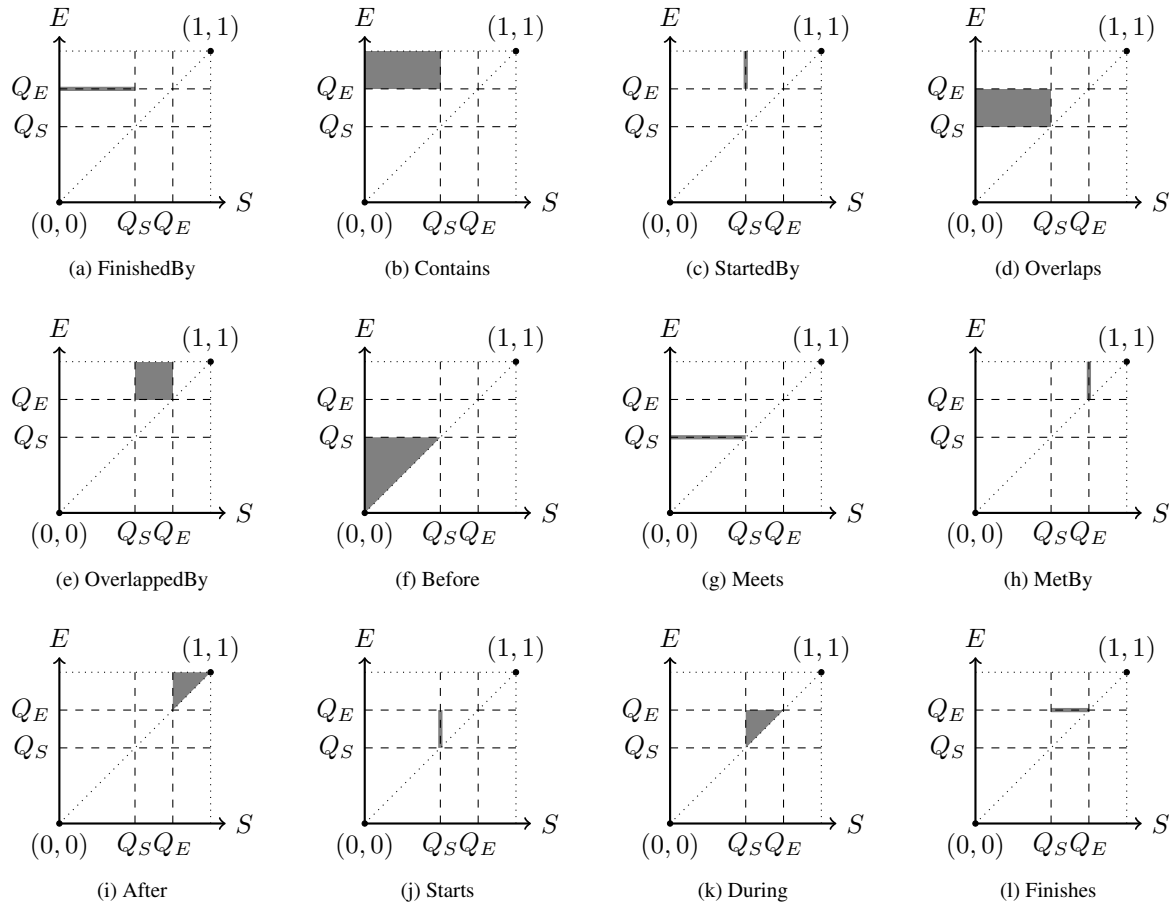
The *Before* and *Meets* query are not affected by *now*-relative query intervals as they only query for elements with certain start-intervals. Despite the fact that the *Finishes*, *FinishedBy*, *Starts*, and *StartedBy* query are not affected by any of the original 8 relationships between objects as shown in Table 1, their conversion is shown in Figure 5 and Table 2 because one may want to define more specific temporal queries.

Example: The example query ends with *current* in dimension d_0. Therefore the end value in d_0 is set to the maximum of the median (0.5) and the current value of *now* (0.7) and the query space is extended. The query matches the current *now* if it contained the current value of *now* in the temporal dimension.

As the query conversion often causes many false-positive results, we also implemented two query alternatives. In the first alternative, whenever the stored *now*-relative data has to be accessed, the query intervals are not expanded but additionally only the affected now-relative elements are queried and both result sets are combined with an logical operator. In the second alternative, the *now*-relative elements are stored in separate indices. As bi-temporal data is investigated, four indices have to be queried in that alternative. One which only contains *now*-relative valid-time data, one which only contains *now*-relative transaction-time data, one which only contains elements which are *now*-relative in both temporal dimensions and one which does not contain any *now*-relative data. Therefore every query is distributed into up to four queries on the separated indices and the results combined by logical operators. As the elements in the underlying B$^+$-tree are ordered the query results could be joined in linear time in both alternatives. Unfortunately, the costs for the additional queries exceed the benefits from much less false-positive results and the alternatives perform worse by up to three orders of magnitude. Therefore we do not investigate these alternatives any further.

Summing up, the complete query conversion process is defined as follows.

1. Identify the type of region queries which are affected by the query (Table 1).
2. Convert the query to the identified region queries in every dimension (Figure 5).
3. Adapt the converted queries if they contain *now*-relative values or if they match the current value of *now* (Table 2).
4. Merge the resulting n two-dimensional region queries to one $2n$ dimensional query.
5. Convert the query to up to $4n$ range queries on the underlying B$^+$-tree (see [7]).



(a) FinishedBy (b) Contains (c) StartedBy (d) Overlaps

(e) OverlappedBy (f) Before (g) Meets (h) MetBy

(i) After (j) Starts (k) During (l) Finishes

Figure 5: 12 queries on intervals. The gray area shows which data is affected by an interval query (Q_S, Q_E) when converted into a two-dimensional region query. The x-axis denotes the value of the starting points, the y-axis denotes the value of the ending points of the intervals. Note that no point lies beneath the diagonal and all points are lying inside $[0, 1]$ in every dimension. The Equals query is a point query on (Q_S, Q_S) and (Q_E, Q_E) respectively (after He et al. [20]).

Example: Finally, the query is converted into four range queries (dimension / range):

d_{0S}	$(0.2, 0.7)$
d_{0E}	$(0.2, 0.7)$
d_{1S}	$(0.3, 0.8)$
d_{1E}	$(0.3, 0.8)$

Berchtold et al. [7] suggest to store all query results in a Point-Set running through it after querying and checking for valid results by matching to the original query. In order to keep the necessary additional space at a minimum the STPA uses the Visitor-Pattern [18]. In order to query the STPA the client has to provide a query region and a query instance. The query instance is a visitor and the STPA a visitable. Thus, for every queried element the given query instance is called by the STPA. The STPA converts the region query into $4n$ range queries and performs them on the underlying B$^+$-tree in the same manner. It provides a start and ending value for the range query and a query object. Every time the B$^+$-tree finds a match for the given query it calls the given query object. This query object then matches the given object against the original query and, if it matches, calls the query object of the original region query. As the different range queries on the B$^+$-tree are independent to each other, the underlying range queries in the STPA may be computed parallel. Figure 6 shows the general structure of the STPA and Figure 7 a sequence of the query system.

Table 2: Incorporating *now*-relative values into range queries. The original one-dimensional query (Q_S, Q_E) is mapped into a two-dimensional region query as shown in Figure 5. The values of the resulting region query are defined as (E_S, E_E) and (S_S, S_E) - a start and an end value for each of both target dimensions. The table lists how these values need to be altered if the query has a *now*-relative end value or if the query matches the current value of *now*. The value c_d is the value of the center point of the pyramids in the dimension d of the converted interval. *Now* is the current value of the ongoing time to which all temporal dimensions are connected.

Query Q	$Q_E = current$	Q matches current *now*
Before	not affected	$E_E = MAX(Q_S, c_d)$
After	$E_S = S_S = MIN(c_d, now)$	$E_S = S_S = MIN(Q_E, c_d)$
Overlaps	$E_E = MAX(c_d, now)$	$E_S = MIN(Q_S, c_d)$ $E_E = MAX(Q_E, c_d)$
OverlappedBy	$E_S = MIN(c_d, now)$	$E_S = MIN(Q_E, c_d)$
During	$S_E = E_E = MAX(c_d, now)$	$E_S = MIN(Q_S, c_d)$ $E_E = MAX(Q_E, c_d)$
Contains	$E_S = MIN(c_d, now)$	$E_S = MIN(Q_E, c_d)$
Starts	$E_E = MAX(c_d, now)$	$E_S = MIN(Q_S, c_d)$ $E_E = MAX(Q_E, c_d)$
StartedBy	$E_S = MIN(c_d, now)$	$E_S = MIN(Q_S, c_d)$ $E_E = MAX(Q_E, c_d)$
Meets	not affected	$E_S = MIN(Q_S, c_d)$ $E_E = MAX(Q_S, c_d)$
MetBy	$S_S = E_S = MIN(c_d, now)$ $S_E = MAX(c_d, now)$	$S_S = MIN(Q_E, c_d)$ $S_E = MAX(Q_E, c_d)$
Finishes	$S_S = E_S = MIN(c_d, now)$ $S_E = E_E = MAX(c_d, now)$	$E_S = MIN(Q_E, c_d)$ $S_E = E_E = MAX(Q_E, c_d)$
FinishedBy	$E_S = MIN(c_d, now)$ $E_E = MAX(c_d, now)$	$E_S = MIN(Q_E, c_d)$ $E_E = MAX(Q_E, c_d)$
Equals	$E_S = MIN(c_d, now)$ $E_E = MAX(c_d, now)$	$E_S = MIN(Q_E, c_d)$ $E_E = MIN(Q_E, c_d)$

3.3 Implementation

The STPA is implemented in the programming language Java, Version 8 and the code is accessible via GitHub [1]. Although it would have been possible, we relinquished on the implementation of a parallel query system like shown in Figure 7. However, this would not have changed the CPU-time or the number of I/Os which are evaluated in the next section.

The underlying file system is abstracted by a `StorageManager` and the file system is only accessible through this `StorageManager` as depicted in Figure 6. This was done for two reasons. First, it allows the user to count the number of I/Os in this `StorageManager` and therefore easily differentiate between I/O operations which are caused by operations on the STPA or from other parts of the application, like the Virtual Machine. Second, the `StorageManager` is exchangeable, for instance if the user wants to switch between an on-disk and an in-memory application or if the user wants to alter the buffer or block size of the abstracted file system.

As we know the properties of the inserted data sets quite well, we did not implement a dynamic approximation of the actual median of the inserted data sets. Instead, it is possible to define the presumable median of the data sets with initialization of the structure. Nonetheless, it is possible to optimize the index structure targeted and bulk-load the whole data set into a clone with an optimized center of the pyramids. The concrete implementation of the competitor, the R^{ST}-tree, is described in the following section.

4 EVALUATION

In this section, the STAP is evaluated and compared to the R^{ST}-tree, since other spatio-temporal methods are able to handle moving-objects data, but not discretely changing spatial data. First the implementation of the

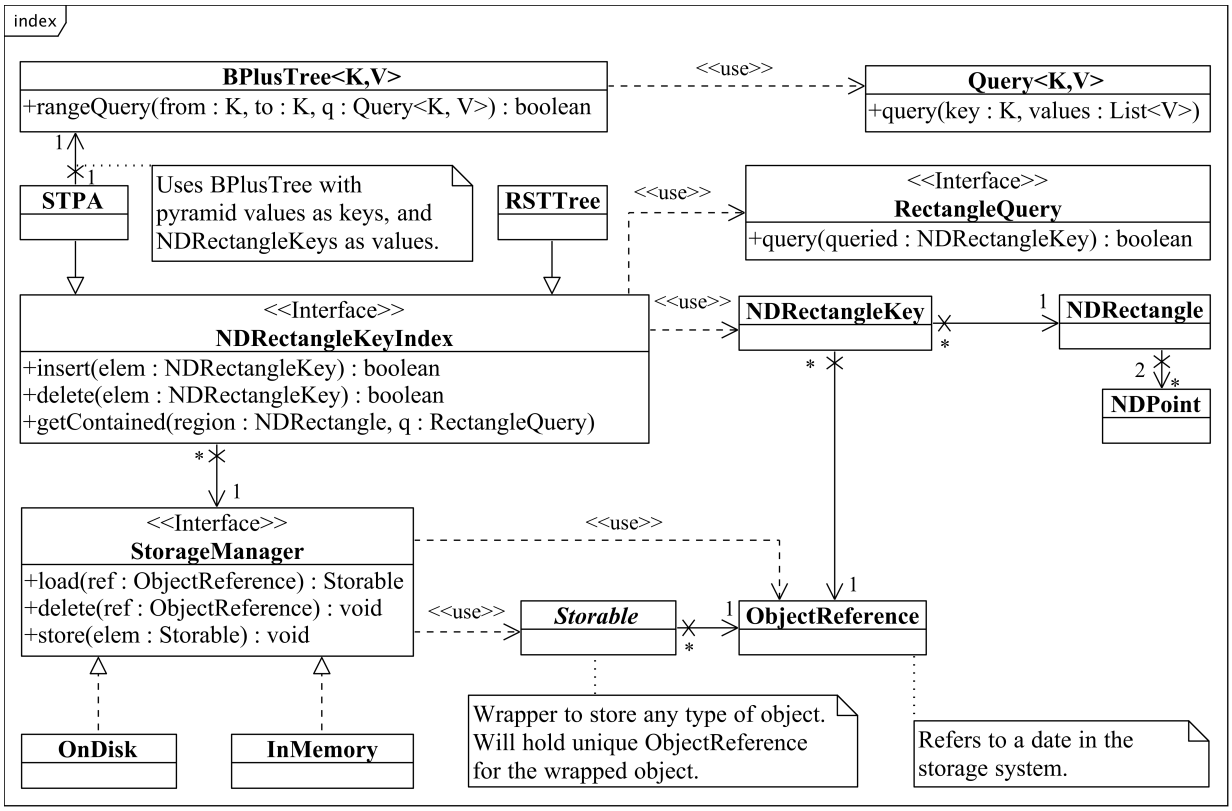


Figure 6: UML class diagram of the STPA. The upper part shows the query system and indexing methods, the lower part shows the structure of the underlying storage manager. Any type of object may be stored, if it has been wrapped into a `Storable`. Every *Storable* is identified by a unique `ObjectReference`.

R^{ST}-tree and the evaluation program are described. Afterwards, the evaluation setup and evaluation results are presented.

4.1 Implementation of the R^{ST}-tree

Our implementation of the R^{ST}-tree uses the previously described `StorageManager`. The performance of the R^{ST}-tree can be adjusted at several positions. First, the parameter $\alpha \in [-1, 1]$ depicts how much the bi-temporal part influences the computation of the volume v of a MBR r by the following formula [34]:

$$v(r) = \begin{cases} bitemp\_area(r)^{1+\alpha} \cdot area(r) & \text{if } \alpha \leq 0 \\ bitemp\_area(r) \cdot area(r)^{1-\alpha} & \text{else.} \end{cases} \quad (1)$$

The margin of a MBR is computed analogously and, since not the absolute margins are crucial for the algorithms of the R^*- and R^{ST}-tree but their correct order, the margins are computed as sums of the lengths of the MBR in every dimension with the above described spatio-temporal weight. The second adjustable parameter is the parametrization value. Every *now*-relative value is replaced by the current value of *now* plus the parametrization value. There are two types of parametrization values. The constant value and the dynamic value. When the dynamic value is chosen, the R^{ST}-tree uses a dynamic time horizon [35]: Every n insertions, the time duration Δt of the last n insertions is computed, where n equates to the number of entries in a node. Then the update interval length is approximated as $UI = \left(\frac{\Delta t}{n}\right) N$, where N is the number of leaf entries and the querying window length as $W = \alpha_W \cdot UI$ with $0 < \alpha_W < 1$. The time horizon H is computed as $H = W + UI$.

The alternate union, splitting, and deletion algorithms are implemented as proposed in the original paper. Summing up, a R^{ST} - tree can be improved by adjusting the α parameter, plus choosing between a constant and dynamic parametrization value and adjusting the α_W parameter, if the latter is chosen.

4.2 Evaluation Program and Setup

The crucial part of the evaluation is the selection of the index-specific parameters and the distribution of the

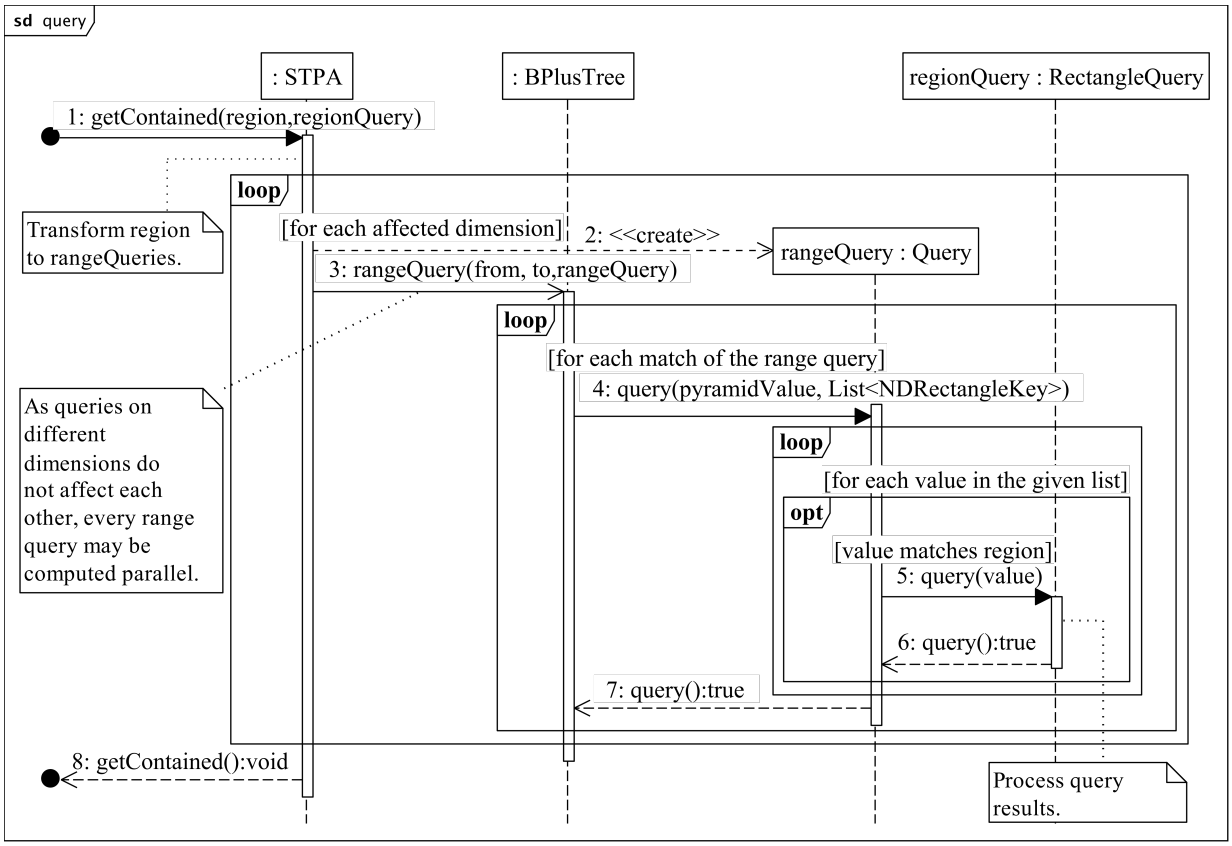


Figure 7: UML sequence diagram of the STPAs parallel query technique. It is assumed, that the original n-dimensional query is converted into $4n$-range queries as described in section 3.2. The structure of the classes used in this diagram is shown in Figure 6.

spatio-temporal data sets. The evaluation program, the parameters and the results are accessible via GitHub [1].

The STPA can only be improved further by altering the center of the pyramids. The center of the pyramids is set to the median of the data set at the beginning of the evaluation, which is almost optimal. Since the temporal values change over time and therefore the approximated median changes over time, the Pyramid Technique, whose center is set to a constant, only provides approximately optimal query results. As in the original paper [7], our experiments show that the benefits in the query cost of providing an optimal center of the pyramids is outweighed by the additional rebuilding cost. Contrary, when using an approximately optimal median, the addition rebuilding cost is negligible [7]. The effect of a not optimized median is investigated in section 4.3.

The R^{ST}-tree can by improved either by using a constant or dynamic parametrization value plus the parameter α. For the workloads described below both types of parametrization values where compared and the constant parametrization showed the best results. Note, that the α and optionally α_W - parameter is adapted anew for each evaluation setup and especially for each spatio-temporal distribution. For the in-memory case the node size can be improved.

One workload is generated as follows. An index is initialized with `initialSize` elements, whose end values on the transaction-time axis are set to *current*. Each of these elements with an *now*-relative end value are considered to be part of an *active* history. A history is a chain of versions of one and the same object which changes its appearance. Within every evaluation step `incSize` elements are inserted, in which a ratio of `startPercentage` elements start a new active history. A ratio of `endPercentage` elements end a still active history by updating the last element in that history with an element whose end value in transaction-time is the current value of *now*. A ratio of `updatePercentage` elements continue an active history by adding an element to a history whose end value is *current* on the transaction-time axis. Adding an element to a history always means that the end value on the transaction-time axis of the latest element is updated to the current value of *now*. Therefore, the start value on

Table 3: Setup parameters for the in-memory and on-disk evaluation plus uniformly distributed, clustered, and skewed data sets.

parameter	value(s)	
`startPercentage`	0.1	
`endPercentage`	0.1	
`updatePercentage`	0.8	
`distribution` `validTimeDistribution`	uniform	$in[0,1]$
	normal	$stdDev = 0.6$ $mean = 0.25$
	skewed normal	$stdDev = 0.3$ $mean = 0.25$ $skew = -1.0$
`vtInfinityProbability`	0.1	
`maxValidTimeLength`	0.1	
`maxElementSize`	0.1	
`incSize`	11000	
`queries`	1000	
`querySize`	0.2	
`dimensions`	$5, 10, 15, 20, 30, 40, 50$	
`StorageManager`	`InMemory`	
	`OnDisk`	4096 bytes per block 0,50 blocks in buffer

the transaction-time axis of the newly inserted element is set to the current value of *now*.

The distribution of the elements in transaction-time is given by the filling process described above. The distribution along the valid-time axis is given by a `validTimeDistribution`, which may either be a uniform, normal or skewed normal distribution. Although it is possible to vary the range and concrete behavior of the distributions, the evaluation only uses data sets which are generated with one of the following settings:

- A uniform distribution in $[0, 1]$.
- A normal distribution with a standard deviation of 0.6 and a mean value of 0.25 in order to create data clusters at 0.25 and most elements lying in $[0, 1]$.
- A skewed normal distribution with a standard deviation of 0.3, a mean value of 0.25, and a skew of -1 in order to achieve a maximal skewness. The skewness is applied on a random number y by $y = (1 - e^{-skew*y})/skew$.

Additionally, elements which lie outside of $[0, 1]$ are discarded. To create *now*-relative valid-time intervals, the parameter `vtInfinityProbability` denotes the likelihood for setting the end time on the valid-time axis of a newly created element to *current*. For all non-*now*-relative elements the parameter `maxValidTimeLength` denotes the maximum length of the valid-time interval. The length is always uniformly distributed in $[0, \texttt{maxValidTimeLength}]$. The start and end values in all non-temporal dimensions are also created with either uniform, normal, or skewed normal distributions. All non-temporal values are created with the same distribution `distribution` and the maximum length of the intervals is given by `maxElementSize`. The number of dimensions is adjustable, but all elements at least have a transaction- and a valid-time dimension. The dimensions are chosen with respect to the long computation time for one workload and in order to show the general behavior of the structures with increasing dimensionality. As the expected impact of the curse of dimensionality should be between 5 and 15 dimensions, more low dimensions are chosen for evaluation. After every insertion of `incSize` elements, the structure is queried by an amount of `queries` in $[0, 1]$ uniformly distributed contained queries. The maximum length of the queries in every dimension is given by `querySize`. This is possible because the valid and transaction-time values are also lying in $[0, 1]$. The current value of *now* starts at 0.25 in order to model that some time already has passed and is increased by a constant for every insert or update operation on the evaluated index. This constant is defined such that the current value of *now* is 0.75 at the end of the workload generation. We also evaluated different block

and buffer sizes, but concentrated on a block size of 4096 byte and a buffer size of 0 due to the fact that a greater buffer size only shifts the results but does not change the general conclusion if the number of inserted elements is big enough. In order to keep the number of elements in one workload at a minimum without falsifying the outcome,the impact of different buffer sizes is not discussed in detail. Keep in mind that the block size is only crucial for the on-disk case, for the in-memory case the maximum number of entries in one node is constant. For our implementation of the R^{ST}-tree, the best maximum node size is 32 entries and the best maximum node size for the underlying B^+-tree of the STPA is 40 entries. The setup of the workload generator is listed in Table 3. We generate 100 workloads for every setup and both, an in-memory and an on-disk working `StorageManager`. Every workload contains 10 evaluation steps and with a initial size of 1000 elements every evaluated index contains 100000 elements at the end of one workload.

In order to provide a reproducible evaluation setup, every pseudo random number generated within one workload depends on a certain random seed. Using this seed, one can reproduce every number as it was generated in a previous generation of a workload with the same set of parameters.

4.3 Results

In this section the evaluation results are presented. For the in-memory case the CPU-time is crucial since both indexing methods do not need much additional memory storage. Note, that the maximum number of entries in the nodes of both structures remains constant for the in-memory case, whereas the node size and not the number of entries remains constant for the on-disk case. The number of I/O operations is the most relevant value for the on-disk case.

For every evaluation setup three diagrams are provided, all generated with Mathematica 10:

- A three-dimensional diagram showing the normalized means for every data point $(size, dimensions)$. This means, that the mean at every data point is divided by the mean of the first data point $(10900, 5)$. By normalizing the plot, we get a better understanding of the relations between increasing size and dimensionality. For a better visibility, only the data from the STPA with a approximately optimized median and the R^{ST}-tree is shown.

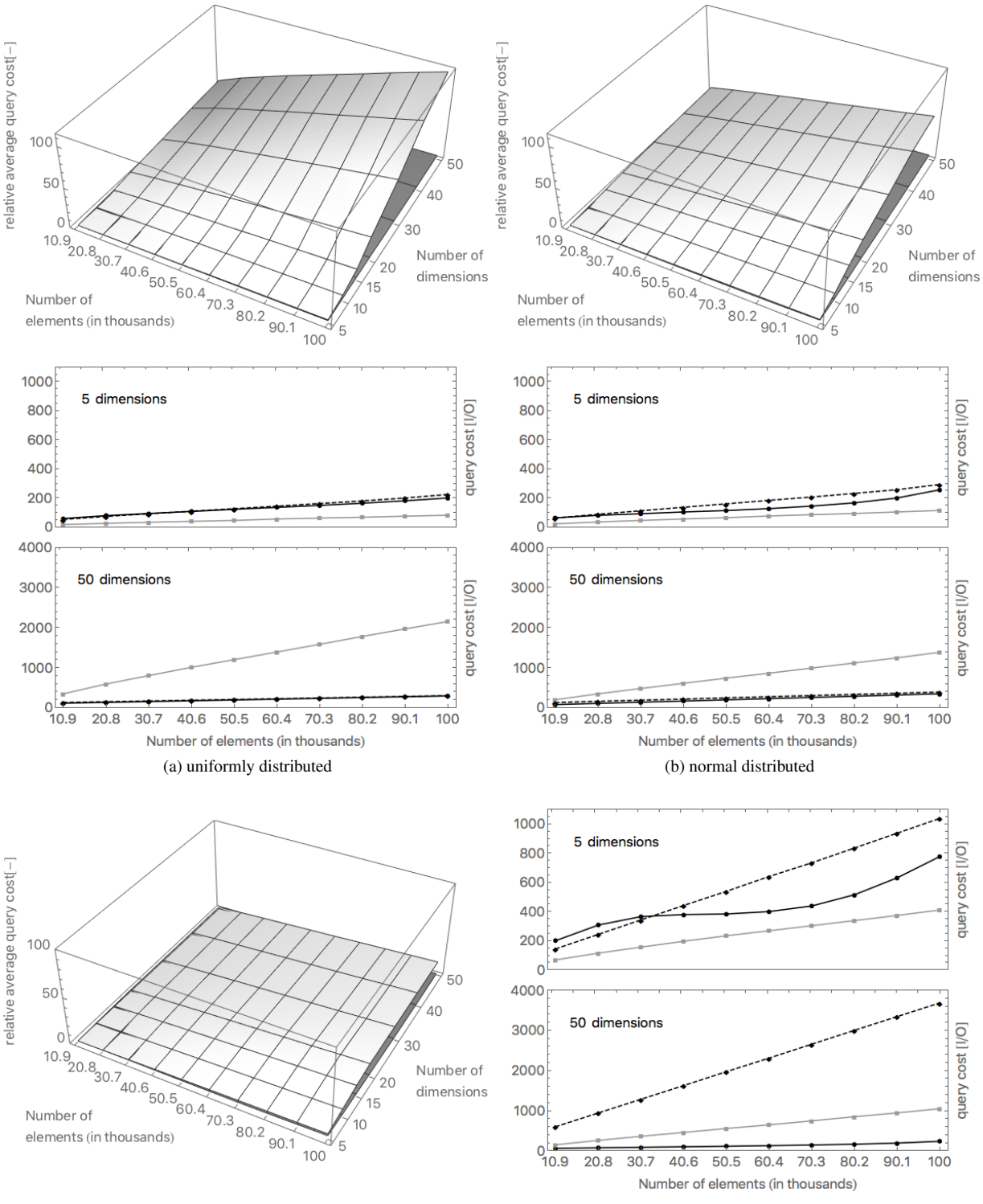
- Two two-dimensional diagrams showing a Line Plot of the absolute mean values for 5 and 50 dimensions.

For both, the in-memory (Figure 9) and the on-disk (Figure 8) case, the R^{ST}-tree has a better query performance than the STPA for lower dimensions, but is clearly outperformed by the STPA for higher dimensions. Taking the trend for an increasing number of elements into account, the STPA is more efficient in higher dimensions.

The better query performance of the STPA in the in-memory case is likely to result from the fact, that for an increasing number of dimensions by a constant maximum number of entries in every node the R^{ST}-tree has to perform an increasing number of floating-point computations when computing the overlaps, margins and volumes of the nodes MBRs. For the on-disk case the performance of the R^{ST}-tree is reduced in higher dimensions, since the maximum number of entries decreases. That is why more nodes have to be created for storing the same number of elements and therefore more nodes have to be accessed when querying. Neither the maximum number of entries in the dir-nodes nor the size of the dir-nodes of the underlying B^+-tree of the STPA is affected by an increasing number of dimensions, because every entry is always identified by a one-dimensional value. However, the maximum number of entries in the leaf-nodes of the underlying B^+-tree is affected by an increasing number of dimensions in the on-disk case and our structure also has to perform more subqueries on high-dimensional data which causes a slightly worse query behavior in the in-memory case.

Through the different distributions the STPA and the R^{ST}-tree show the same general behavior but especially the impact of the median in the STPA variates. For uniformly and normal distributed data choosing the default median of 0.5 seems to have less impact on the evaluation results as the STPA with the default median is only slightly worse than the STPA with approximately optimized median. For the in-memory case (Figure 9) the mean values seem to be better for the configuration with the default median, but as the results for optimized and default median also vary about 0.4 we cannot make a general conclusion on the impact of the median. Such an impact can clearly be seen for the skewed distribution. There, the approximately optimized median shows clearly better results than the default median setting. The dent in the plot for the skewed distribution in lower dimensions (Figures 8c, 9c) may be caused by the greater impact both time dimensions. Especially if it is recalled, that the time moved from 0.25 to 0.75 through one evaluation.

With respect to the 3D-plots, which show the relative means of the STPA with approximately optimized medians and the R^{ST}-tree, the relative difference is larger for uniformly, less for normal, and even lesser for skewed distributed data. This means, that the increase of the

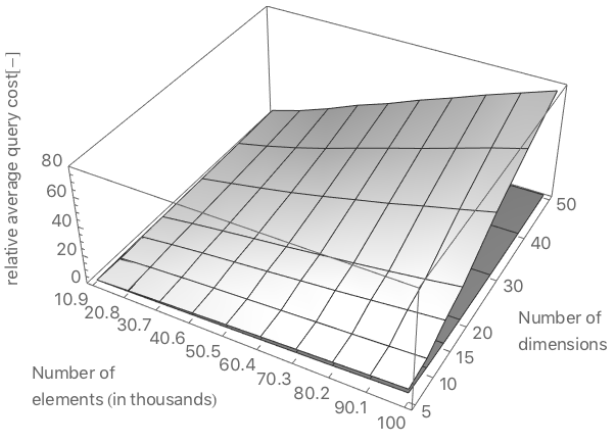


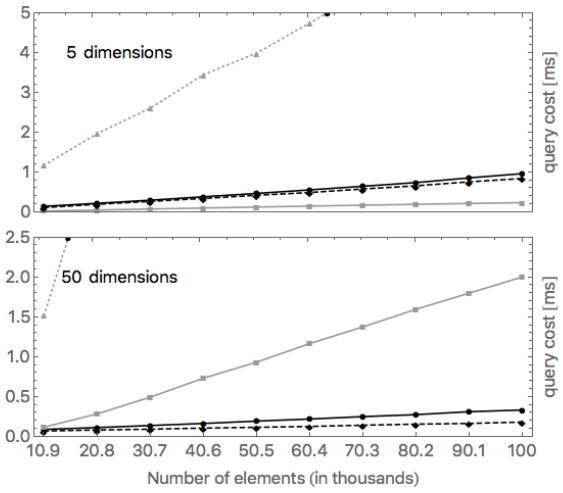
(a) uniformly distributed

(b) normal distributed

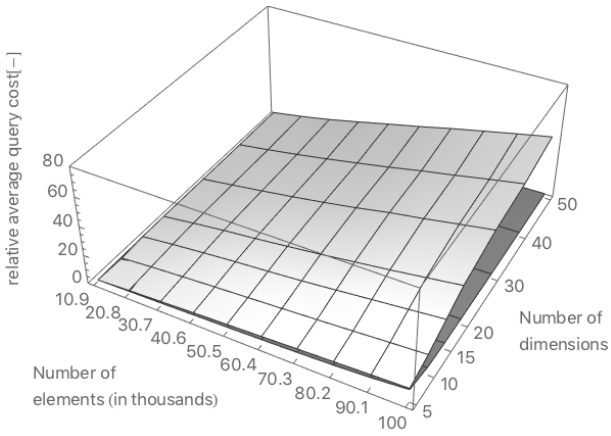
(c) skewed distributed

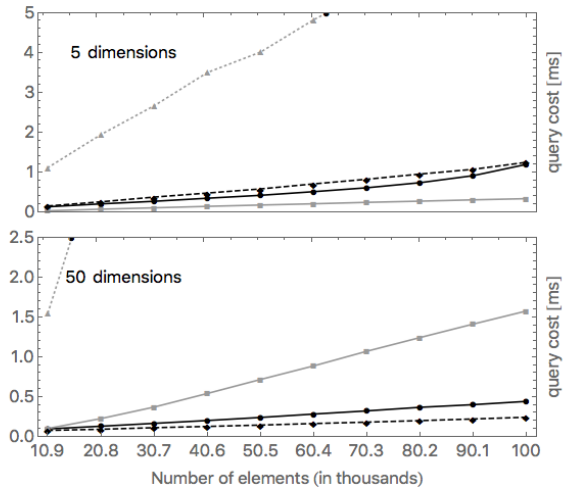
Figure 8: Comparison of the STPA with approximated optimal median (black), default median (dashed) and the R^{ST}-tree (gray) in the on-disk case. The absolute query cost of the sequential scan always equals the number of elements.



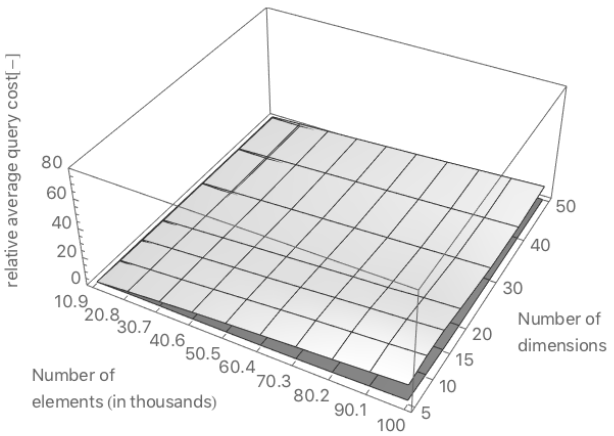


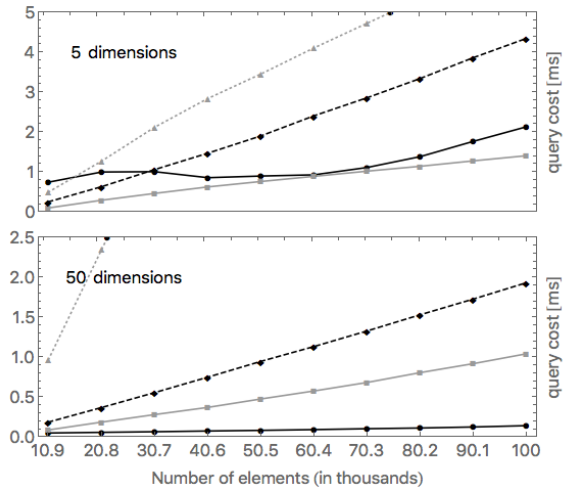
(a) uniformly distributed





(b) normal distributed





(c) skewed distributed

Figure 9: Comparison of the STPA with approximated optimal median (black), default median (dashed) and the R^{ST}-tree (gray) in the in-memory case. The absolute query cost of the sequential scan (gray dotted), ranges from 1.2 to 16 and is not shown fully for a better comparability between the other methods.

query cost in the R^{ST}-tree depends on the distribution of the data, whereas the STPA seems to have a nearly equal increase for all three distribution types.

We observe a similar behavior for the costs of insertion, update, and deletion in the indexing methods. In difference to the query cost the STPA outperforms the R^{ST}-tree also in absolute terms and lower dimensions. We do not present these results in detail here as the query cost is the crucial value for comparing the indexing methods.

5 Conclusions and Outlook

We presented a new indexing method for high-dimensional spatio-bitemporal data with a discretely changing spatial extend, the Spatio-Temporal Pyramid Adapter (STPA), which is based on the Extended Pyramid Technique [7]. The evaluation shows, that the overhead generated by the conversion of an n-dimensional spatio-temporal query to up to $4n$ one-dimensional queries is outweighed by the benefits of the Pyramid Technique in higher dimensions (> 10). The STPA is not suitable for continously moving objects and the configuration of the medians is crucial for distributions with a high skewness. With respect to that, an approximately optimal median as generated by the Extended Pyramid Technique is sufficient. For distributions with more than one cluster, the P^+-tree [47] may be used instead of the Extended Pyramid Technique.

The STPA can be implemented for parallel computation and is not restricted to a certain number of temporal dimensions. As it is based on the B^+-tree it may be implemented on top of common database systems.

Following the outcome of this paper, we see several research directions. First, the implementation of the P^+-tree [47] as basis of the STPA may be worthwhile, as such a index may handle data with more than one cluster better. Also, the impact of the ongoing time with respect to the median may be investigated further, as the evaluation indicated that at there is an impact at least for skewed distributions.

Second, another good way to access high-dimensional spatio-temporal data may be a combination of the X-tree [8] and the R^{ST}-tree [34], as they both extend the R^*-tree and their algorithms independently address the special requirements of high-dimensional and spatio-temporal data respectively. As the main aspect of the X-tree is the subtle usage of the main memory in order to achieve fewer disk accesses, it is not useful for our in-memory case and therefore this combination of X- and R^{ST}-tree was not further investigated.

Third and last, we plan to extend our evaluation program, which is based on the generation and measurement of clearly defined workloads, to a more general performance test system. Therefore, we need to create an automated system, which does not rely on test cases which where foreseen by an expert, but derives the test cases from the specification.

6 Acknowledgements

The funding of the researchers group FOR-1546 by the German Research Foundation (DFG) is acknowledged.

References

[1] Source code, parameter sets, and results on GitHub. [Online]. Available: https://github.com/mmenning/sthd_indexing.git

[2] T. Abraham and J. F. Roddick, "Survey of Spatio-Temporal Databases," *GeoInformatica*, vol. 3, no. 1, pp. 61–99, 1999.

[3] J. F. Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[4] P. D. R. Bayer and D. E. M. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "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*. ACM, 1990, pp. 322–331.

[6] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzwicz, "Sumo (simulation of urban mobility)," in *Proceedings of the 3rd International Conference on Advances in System Simulation*, Oct. 2011, pp. 55–60.

[7] S. Berchtold, C. Böhm, and H.-P. Kriegel, "The Pyramid-technique: Towards Breaking the Curse of Dimensionality," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. ACM, 1998, pp. 142–153.

[8] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," in *Proceedings of the 22. International Conference on Very Large Databases*. Morgan Kaufmann, 1996, pp. 28–39.

[9] M. Breunig, E. Rank, M. Schilcher, A. Bormann, S. Hinz, R. P. Mundani, Y. Ji, M. Menninghaus, A. Donaubauer, and H. Steuer, "Towards Computer-Aided Collaborative Subway

Track Planning in Multi-Scale 3D City and Building Models," in *Proceedings of the 6th 3D geoinfo conference*, 2011, pp. 17–25.

[10] T. Brinkhoff, "A Framework for Generating Network-Based Moving Objects," *GeoInformatica*, vol. 6, no. 2, pp. 153–180, Jun. 2002.

[11] E. Butwilowski, A. Thomsen, M. Breunig, P. V. Kuper, and M. Al-Doori, "Modeling and Managing Topology for 3-D Track Planning Applications," in *3D Geoinformation Science*. Springer International Publishing, 2015, pp. 37–53.

[12] S. Chen, C. S. Jensen, and D. Lin, "A Benchmark for Evaluating Moving Object Indexes," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1574–1585, Aug. 2008.

[13] J. Clifford, C. Dyreson, T. a. s. Isakowitz, C. S. Jensen, and R. T. Snodgrass, "On the Semantics of Now in Databases," *ACM Transactions on Database Systems*, vol. 22, no. 2, pp. 171–214, Jun. 1997.

[14] D. Comer, "The Ubiquitous B-Tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, Jun. 1979.

[15] C. Düntgen, T. Behr, and R. H. Güting, "BerlinMOD: A Benchmark For Moving Object Databases," *The VLDB Journal*, vol. 19, no. 5, pp. 687–714, 2010.

[16] M. Egenhofer, "A Formal Definition of Binary Topological Relationships," in *Foundations of Data Organization and Algorithms*. Springer Berlin Heidelberg, 1989, pp. 457–472.

[17] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys." *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, ser. Elements of Reusable Object-Oriented Software. Pearson Education, 1994.

[19] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. ACM, 1984, pp. 47–57.

[20] Z. He, M.-J. Kraak, O. Huisman, X. Ma, and J. Xiao, "Parallel indexing technique for spatio-temporal data," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 78, no. 0, pp. 116–128, Apr. 2013.

[21] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An Adaptive B+-Tree Based Indexing Method for Nearest Neighbor Search," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 364–397, Jun. 2005.

[22] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," in *Proceedings of the 13. international conference on Very Large Data Bases*. VLDB Endowment, 2004, pp. 768–779.

[23] V. Köppen, M. Schäler, and R. Schröter, "Toward Variability Management to Tailor High Dimensional Index Implementations," in *Proceedings of the IEEE Eighth International Conference on Research Challenges in Information Science*. IEEE, May 2014, pp. 1–6.

[24] H.-P. Kriegel, M. Pötke, and T. Seidl, "Interval Sequences: An Object-Relational Approach to Manage Spatial Data," in *Advances in Spatial and Temporal Databases*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 481–501.

[25] K.-I. Lin, H. V. Jagadish, and C. Faloutsos, "The TV-tree: An Index Structure for High-Dimensional data," *VLDB Journal*, vol. 3, no. 4, pp. 517–542, 1994.

[26] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-Temporal Access Methods," *IEEE Data Engineering Bulletin*, vol. 26, no. 2, pp. 40–49, 2003.

[27] M. A. Nascimento and J. R. O. Silva, "Towards Historical R-trees," in *Proceedings of the 1998 ACM symposium on Applied Computing*. New York, NY, USA: ACM, 1998, pp. 235–240.

[28] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis, "Evaluation of Access Structures for Discretely Moving Points," in *Proceedings of the International Workshop on Spatio-Temporal Database Management*. London, UK: Springer Berlin Heidelberg, 1999, pp. 171–189.

[29] L. V. Nguyen-Dinh, W. G. Aref, and M. Mokbel, "Spatio-Temporal Access Methods: Part 2 (2003-2010)," *IEEE Data Engineering Bulletin*, vol. 33, no. 2, pp. 46–55, 2010.

[30] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Transactions on Database Systems*, vol. 9, no. 1, pp. 38–71, Mar. 1984.

[31] N. Pelekis, B. Theodoulidis, I. Kopanakis, and Y. Theodoridis, "Literature review of spatio-temporal database models," *The Knowledge Engineering Review*, vol. 19, no. 03, pp. 235–274, Sep. 2004.

[32] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," in *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1981, pp. 10–18.

[33] J.-M. Saglio and J. Moreira, "Oporto: A Realistic Scenario Generator for Moving Objects," *GeoInformatica*, vol. 5, no. 1, pp. 71–93, Mar. 2001.

[34] S. Saltenis and C. S. Jensen, "R-tree Based Indexing of General Spatio-Temporal Data," Tech. Rep., 1999.

[35] S. Saltenis and C. S. Jensen, "Indexing of moving objects for location-based services," *Proceedings of the 18th International Conference on Data Engineering*, pp. 463–472, 2002.

[36] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2000, pp. 24–342.

[37] M. Schäler, A. Grebhahn, R. Schröter, S. Schulze, V. Köppen, and G. Saake, "QuEval: Beyond high-dimensional indexing à la carte," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1654–1665, 2013.

[38] R. T. Snodgrass, "Temporal databases," *IEEE Computer*, vol. 19, pp. 35–42, 1986.

[39] B. Stantic, R. Topor, J. Terry, and A. Sattar, "Advanced Indexing Technique for Temporal Data," *Computer Science and Information Systems*, vol. 7, no. 4, pp. 679–703, 2010.

[40] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento, "On the Generation of Spatiotemporal Datasets," in *Proceedings of the 6th International Symposium on Advances in Spatial Databases*. London, UK: Springer-Verlag, 1999, pp. 147–164.

[41] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos, "Overlapping Linear Quadtrees: a Spatio-temporal Access Method," in *Proceedings of the 6th ACM international symposium on Advances in geographic information systems*. New York, NY, USA: ACM, 1998, pp. 1–7.

[42] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos, "On the Generation of Time-Evolving Regional Data," *GeoInformatica*, vol. 6, no. 3, pp. 207–231, Sep. 2002.

[43] P. Van Oosterom, W. Quak, and T. Tjissen, "Testing current DBMS products with real spatial data," in *Proceedings of 23rd Urban Data Management Symposium*, Prague, Czech Republic, 2002, pp. VII.1–VII.18.

[44] J. Wang, J. Lu, Z. Fang, T. Ge, and C. Chen, "PL-Tree: An Efficient Indexing Method for High-Dimensional Data," in *Advances in Spatial and Temporal Databases*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 183–200.

[45] R. Weber, H. J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," in *Proceedings of the 24rd International Conference on Very Large Data Bases*, 1998, pp. 194–205.

[46] M. F. Worboys, "A Unified Model for Spatial and Temporal Information," *The Computer Journal*, vol. 37, no. 1, pp. 26–34, 1994.

[47] R. Zhang, B. C. Ooi, and K. L. Tan, "Making the Pyramid Technique Robust to Query Types and Workloads," in *Proceedings of the 12th International Conference on Data Engineering*. IEEE Comput. Soc, Mar. 2004, pp. 313–324.

AUTHOR BIOGRAPHIES

Mathias Menninghaus received his diploma in applied system sciences from the University of Osnabrück, Germany, in 2011. He was a research assistant at the Karlsruhe Institute of Technology, Germany, in 2011 and 2012 and is currently research assistant at the University of Osnabrück. His research interests are: spatio-temporal databases and automatic performance test generation.

Dr. Martin Breunig received his diploma in computer science from the Technical University of Darmstadt, Germany, and his PhD and habilitation, respectively, from University of Bonn, Germany. He is chair of Geoinformatics at Karlsruhe Institute of Technology (KIT). He had the leadership of several interdisciplinary joint projects within the Geotechnologies program founded by the German Ministry of Education and Research and the German Research Foundation (DFG). He is also leading the DFG research group "Cooperative Planning for Multi Scale 3D City- and Building Models". Martin Breunig has been working as guest professor at ETH Zürich, University of Wien, Staffordshire University, ENSG Nancy, and American University in Dubai. His main research interests include Geographic Information Systems (GIS), geo-databases, 3D/4D GIS, and mobile GIS.

Prof. Dr.-Ing. Elke Pulvermüller is a professor in the Department of Mathematics and Computer Science at the University of Osnabrueck (Germany). There, she is head of the Software Engineering research group. Previous to her appointment at Osnabrueck she has been a senior researcher / research assistant at the University of Luxembourg (2006 - 2007), at the Friedrich Schiller-University of Jena (Germany) and at the Universitaet Karlsruhe (Germany). She received her doctoral degree from the Friedrich Schiller-University of Jena in 2006. Her research focuses on new approaches in software and quality engineering. Elke Pulvermueller is a member of the German Computer Society (GI) and the ACM.