A Framework of a Generic Index for Spatio-Temporal Data in Concert

Lukas Relly[1,2], Alexander Kuckelberg[1], and Hans-J. Schek[1]

[1] Database Research Group
Institute of Information Systems, ETH Zürich
CH-8092 Zürich, Switzerland
{relly, kuckelbe, schek}@inf.ethz.ch
http://www-dbs.ethz.ch
[2] Current address: UBS AG
Postfach, 8098 Zürich
lukas.relly@ubs.com

Abstract. In this paper we present the prototype database system CONCERT and the incorporation of a framework of a generic index tree for spatio-temporal data. We show the ideas behind the CONCERT architecture as far as they are important to understand the framework approach presented. We show how the index is based on the conceptual behaviour of data in contrast to generalized algorithms or methods. Because of the simplicity of R-trees we take an R-tree like structure to explain our generic spatio-temporal index. It is remarkable that in CONCERT a generic index can be defined without any predefined "hard-wired" spatial or temporal data types such as intervals or rectangles. As it turns out the only important properties needed are an `OVERLAP` and a `SPLIT` function, the first one checking for spatial or temporal overlap of objects, the second one providing a hierarchical decomposition of the data space into subspaces. If, in addition, splitting of data objects is allowed we are able to define manageable node sizes, leading to an improved generic index similar to R+-trees or other derivations.

1 Introduction

Spatio-temporal database systems have obtained increasingly high attention due to the fact of more spatial and temporal data being available and complex application requirements demanding an integrated view of space and time in large data collections. Traditional database systems and database technology is inadequate to fulfill these requests. Spatio-temporal applications have many special requirements. They deal with very complex objects, for example objects with complex boundaries such as clouds and moving points through 3D space, large objects such as remote sensing data, or large time series data. These complex objects are manipulated in even more complex ways. Analysis and evaluation programs draw conclusions combining many different data source. Therefore new techniques have to be found and well-established techniques from spatial only and temporal only databases have to be integrated. This integration relies upon a flexible and open system architecture allowing the combination of different — and so far studied isolated from each other — techniques of query processing, query optimization, indexing, and transaction management.

In order to better support advanced applications, the standardization effort of SQL3 specifies, among others, new data types and new type constructors. Most recently, SQL3 and object-orientation have fostered the development of generic extensions called datablades [12], cartridges [13], or extenders [10]. They are based on the concept of abstract data types and often come with specialized indexing. The important idea behind is to provide a generic system capable of being extended internally by application-specific modules integrating the functionality required for specialized applications as close as possible into the DBMS taking advantage of the monolithic architecture while avoiding its deficiencies. Figure 1 illustrates the general system architecture of extensible systems.

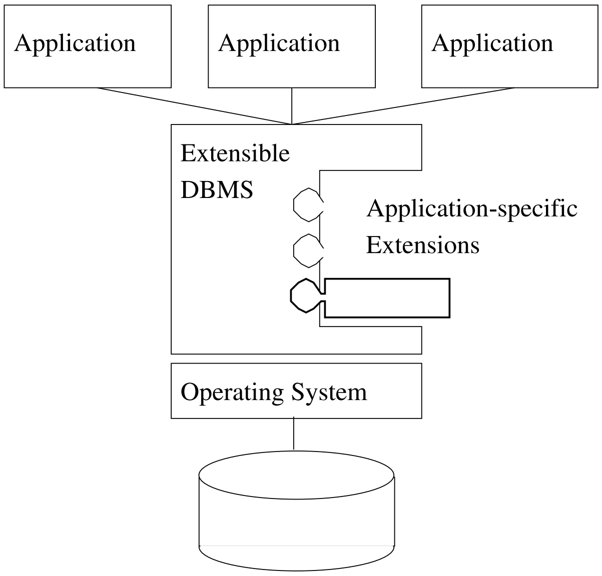

Fig. 1. The extensible system architecture

In this paper we will present the architecture of the database prototype system Concert, that follows the extensible system architecture. We show the ideas behind Concert and how this approach, which is based on the characterization of the behaviour of data and not on a type system, can be used to incorporate generic spatio-temporal indexes into the kernel system. The index together with the Concert kernel system forms a framework (in the sense of [5] or [11]) for spatio-temporal data management systems in which each limitation to "hard-wired" data types is avoided. We see this as the main contribution of the paper. The advantage of such an approach is that a generic R-tree like structure can be adapted to many variants of actual implementations of trees indexing spatio or temporal data or both just differing the functions `OVERLAP` and `SPLIT`.

1.1 Related Work

In [21], Stonebraker introduced the idea of a generic B-tree that depends only on the existence of an ordering operation to index arbitrary data objects. This idea is consequently extended in the Concert kernel system which identifies data by its behaviour and not by its type. In our own previous work [17, 18] we presented algorithms for generic inverted file indexes, generic vertical partitioning, and simple generic spatial partitioning useful for raster data organization. The generic index presented here is closely related to generic search tree approaches such as GiST [9] with one main difference: GiST expects the implementation of certain generic methods for different key values data types (e.g. $\mathbb{Z}$ for B-trees or $\mathbb{R} \times \mathbb{R}$ for R-trees). In contrast Concert and our generic index is based on the classification of data behavior. The concrete data type is only published to the Concert system and all other extensions just use its classification but no knowledge about the data itself. However, the ideas in [9] are complementary and can easily be integrated into the generic index we present in this paper.

In the following Section 2 we first repeat some basic notions, ideas and architectural issues of the Concert system which are necessary to understand the generic index tree for spatio-temporal data which is presented in Section 3. We also discuss the generic algorithms for the construction and usage of the generic indexing structure in Section 3 before we conclude in Section 4.

2 The Concert Architecture

The Concert prototype database system's primary focus is to provide a platform for investigation of physical database design in an extensible kernel system as well as for generic query processing over heterogeneous distributed repositories [2, 17]. The Concert general architecture is similar to other extensible DBMS such as Postgres [16, 20], DASDBS [19], or Starburst [8]. It consists of a kernel system for low-level data management and query processing as well as an object manager providing advanced query functionality. The kernel is responsible for the management of individual objects within collections, resource and storage management and low-level transaction management. Concert provides a generic framework ([5]) for physical database design for external objects. The object manager combines different collections adding join capability to the kernel collections. Figure 2 shows an overview of the Concert architecture.

One of the important issues of the Concert architecture is it's built-in support for interoperability. Todays DBMS make the implicit assumption that their services are provided only to data stored inside the database. All data has to be imported into and being "owned" by the DBMS in a format determined by

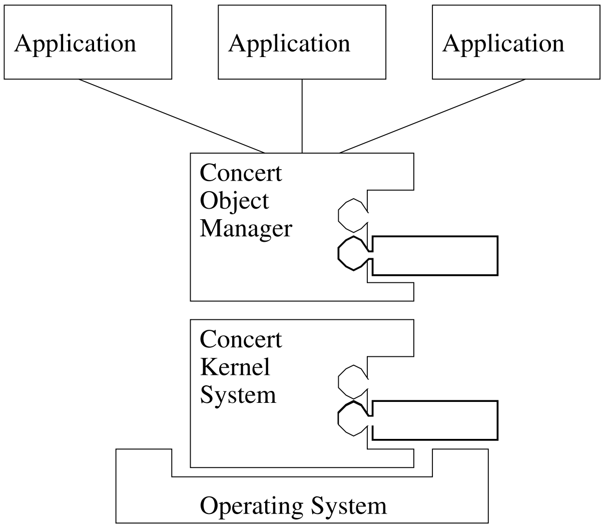

Fig. 2. Overview of the CONCERT system architecture

the DBMS. Traditional database applications such as banking usually meet this assumption. These applications are well supported by the DBMS data model, its query and data manipulation language and its transaction management. Advanced applications such as spatio-temporal applications however differ in many aspects from traditional database applications. Individual operations in these applications are much more complex and not easily expressible in existing query languages. Powerful specialized systems, tools and algorithms exist for a large variety of tasks requiring a spatio-temporal database system to interoperate with other systems. CONCERT's mechanism of providing the necessary infrastructure for application extensions can be used to interoperate with other systems. According to its system architecture consisting of a kernel system and an object manager layer, application extensions and interoperability can take place on both levels. The kernel is responsible for managing and performing physical database design for individual objects and allows access to individual remote data objects. The object manager handles collections of objects and combine them using a query language. In the same way as it combines *internal* collections, it can combine *external* collections made accessible through application-specific wrappers being plugged into the object manager. Details can be found in [15].

Throughout the rest of the paper, we illustrate how the CONCERT extensible architecture can be exploited for a flexible spatio-temporal DBMS. We particularly focus on a generic R-tree-like indexing mechanism but other indexing structures than trees might also be possible.

2.1 The Concert Kernel System

The CONCERT kernel system consists of two components, the Storage Object Manager (SOM) and the Abstract Object Manager (AOM). The SOM provides standard DBMS base services such as segment management, buffer management, and management of objects on database pages. It is tightly integrated into the underlying operating system exploiting multi-threading and multiprocessing. It uses a hardware-supported buffer management exploiting the operating systems virtual memory management by implementing a memory-mapped buffer [1]. Details of the CONCERT SOM can be found in [14].

The Abstract Object Manager provides the core functionality for extending the kernel with application-specific code. It implements a fundamental framework for the management of collections. It uses the individual objects handled internally by the SOM or, through its interoperability capability, by an external storage system, combining them into collections. The AOM collection interface defines operations to insert, delete and update individual objects. Retrieval of objects is performed through direct object access or through collection scans. Scans are initiated using an optional predicate filtering the qualifying objects. Figure 3 shows the (simplified) collection interface definition.

```
createInstance (coll_description) :- coll_handle
deleteInstance (coll_handle)

insertObject   (coll_handle, object) :- object_key
deleteObject   (coll_handle, object_key)
updateObject   (coll_handle, old_key, object) :- new_key
getObject      (coll_handle, object_key) :- object

scanStart      (coll_handle, predicate) :- scan_handle
scanGet        (scan_handle) :- object
scanNext       (scan_handle)
scanClose      (scan_handle)
```

Fig. 3. AOM collection interface (simplified)

Indexes are treated similar to base collections. From the AOM's perspective, they are collections of objects containing the index entry (the attribute to be indexed on) and the object key of the base collection. Accessing data using an index is performed by an index scan identifying all qualifying object keys that are subsequently used to retrieve the objects themselves. Depending on the kind of the index and the query, the objects retrieved might have to be filtered further performing a false drops elimination.

On object insertion, the object is inserted into the base collection first. The insertion operation of the base collection returns an object key that can be used for the index entry. Because the index uses the same interface as the base collection, hierarchies of indexes can easily be built. This allows for example an inverted file index to be indexed itself by a B-tree index. The conceptual equality of base collections of objects and the index collections of their keys enables the components to be combined in many different ways providing a powerful tool that goes fare beyond simple indexing and includes things such as constraint validation and trigger mechanisms.

A special aspect of the CONCERT approach is the fact, that indexing in particular and physical database design in general is performed for abstract data types. Obviously it is not possible to index completely unknown objects. Some knowledge of the user-defined objects has to be available to the storage system. In the following section, we address this issue identifying a small set of physical design concepts sufficient to allow most physical design decisions, specially R-tree like generic index frameworks with minimal restrictions to the flexibility of external objects.

2.2 Extensible Physical Database Design in the Concert Kernel

To explain the CONCERT physical database design extensibility idea, let us first look at a standard B-tree index [4], as it is implemented in most database systems. A B-tree stores keys of objects ordered by their values. The ordering operation is chosen depending on the data type of the key value: for data type "NUMBER", standard ordering over numbers is used whereas "STRING" data types are lexicographical ordered. Another ordering operation is used for "DATE" data types. Much more data types can be possible but the central aspect when data is indexed by an B-tree is that there is an attribute which can be ordered (see [21]). On the other hand data types which can't be ordered can't be indexed by a B-tree.

The generalization of this observation is the basis of the *abstract object storage type* concept in CONCERT: The knowledge of the type of data is not needed to manage it but its conceptual behaviour and the operations associated with these concepts, called *concept typical operations*[1]. Four concepts data type might follow can be identified:

- `SCALAR`: A data type belongs to the `SCALAR` concept, if there is an ordering operation. All data types behaving *like* a `SCALAR` can be indexed e.g. by a B-tree.
- `LIST`: A data type belongs to the `LIST` concept, if it consists of a set of components over which an iterator is defined. A data type behaving like a `LIST` might be a list of keywords, forming the inverted index of a text document. The concept typical operations are `FIRST` and `NEXT`.
- `RECORD`: A data type belongs to the `RECORD` concept if it is a ordered concatenation of components which themselves are arbitrary concepts. The concept

[1] By concept typical operations, we mean the methods the user must provide to allow CONCERT to interpret a given abstract object storage type. Concept typical operations are the means of extensibility of the CONCERT kernel.

typical operation is the SUB_OBJECT operation returning a component of the RECORD. A RECORD is the realization of object partitioning.
- SPATIAL: A data type belongs to the SPATIAL concept if it is extended in an arbitrary data space and can be indexed with respect to its spatial properties. This is the most important concept in the context of spatio-temporal data and will be explained in detail in the next subsection. Note that this concept is not limited to geometric space or time resp. the corresponding data types. The concept typical operations are OVERLAPS, SPLIT, COMPOSE and APPROX which are also explained in the following.

Spatial Indexes and the SPATIAL-Concept: Even if the SPATIAL-concept is not limited to geometric space or time data, these are the most common data spaces of this concept.

There exists a multitude of spatial and temporal index structures in the literature. Gaede and Günther give an overview of important multidimensional access methods in [6] and a comparison of access methods for temporal data can be found in [22].

In order to come up with a generic spatial-temporal index we observe that most spatial, temporal and spatio-temporal indexes are similar in the following way: they organize the objects according to their space-subspace-relationships. Typical queries that are well supported with spatial and temporal indexes include the search for objects in certain regions using spatial predicates (overlaps, intersects, covers) or objects concerning certain time intervals. In order to support such queries, most spatial indexes decompose the data space into — possibly overlapping — subspaces assigning these subspaces to subtrees of the index structure. Queries are well supported, if only a few subtrees have to be searched. Therefore, physical clustering of spatially and temporally neighboured objects is essential. Indexing structures other than trees are possible but are not discussed in this paper. The important point here is that indexing structures can be implemented and plugged into CONCERT independent of the data type indexed by just using the concept typical operations of data objects following the SPATIAL concept.

We found that there are three important spatial classes of objects involved in spatial indexing. These classes are closely related to each other:

- Application objects: In the context of extensible systems, these objects are abstract and might vary from one application area to the other.
- Query objects: These objects are used to formulate queries. Depending on the application, the type of these objects might be the same as for application objects. However, usually, only a small set of different object types are used for querying, the most important being the rectangular query window.
- Space covering objects: These spatial objects represent the space covered by a node or a subtree in the index structure. Typical index structures use rectangles for their subspace describing objects. Other more complex objects such as convex polygons are possible, but in order to keep index traversing operations efficient, simple geometric objects usually are preferred.

Note that at this point we are much more general and flexible that given implementations of trees: We only determine the concept of the node objects, not their structure or type.

In traditional systems, the third class of spatial objects is predefined by the index method provided and therefore "hard-wired" in the system, e.g. a space covering rectangle in the R-tree. Therefore, also the first two classes have to be predefined by the DBMS requiring the user to convert his or her data into the DBMS format. This clearly is not desirable in the context of extensible systems. CONCERT follows a different paradigm. Rather than predefining spatial index objects and thereby forcing the data structure of user objects and user queries, it allows the user to implement not only the application and query objects, but also corresponding index objects. The spatial index algorithm is written in a generic way only exploiting the spatial space-subspace relationships through method invocation.

These spatial[2] relationships define the concept typical operations of the `SPATIAL` concept as follows:

- The operation `OVERLAPS` checks for spatial overlap of two objects or data spaces. It is used by index tree navigation to prune the search space. It also helps finding the appropriate leaf node on index insertion, as we will explain in more detail in Section 3. It therefore is not defined as a simple predicate operation, but rather as an operation returning an integer value indicating the degree of overlap, while negative values are used as a measure for indicating the distance between two objects[3]. Finally, the `OVERLAPS` operation is used to approximate spatial and temporal overlaps, intersects and covers predicates.
- The `SPLIT` operation divides a spatial object into several (smaller) spatial objects. This operation can be used to a priori divide large user objects into smaller ones before inserting them or while reorganizing the search tree. It is also used to split index nodes when they become too large. The question when nodes are splitted and which concrete spatial object is returned is determined by the implementation of the `SPLIT`-operation of the associated data type following the `SPATIAL` concept. In this way, a split-operation can return a simple set of rectangles resulting from splitting a "large" rectangle (like in R+-trees) but also a set of complex polygons or spheres.
  Here the terms small and large are used in two contexts. Large objects can be objects occupying a lot of storage space, as well as objects covering a large data space. In both cases, it might be beneficial to divide a large object into several smaller ones. In the first case, memory allocation and

[2] Note, that with spatial, we do not restrict the objects to geometric space. The statements made are valid also for temporal space and other data spaces.

[3] Note, that there is no restriction about the implementation of these operations. Programmers might decide to only distinguish two values — overlapping and non-overlapping — for the `OVERLAPS` operation. The concepts described here will work in the same way, however, the optimizations described are not possible.

buffering is easier, in the second case, the data space in index nodes is smaller allowing a better pruning in the index structure. The behaviour depends on the implementation of the `SPLIT`-operation.

- The `COMPOSE` operation recombines spatial objects that have previously been splitted to an "overall" spatial object. This operation is used for the reconstruction of large data objects which have been splitted on insertion, as well as for index node description objects when index nodes are merged. Note that we make no assumption about the implementation of the operations for a given data type but only the conceptual behaviour of the data is important. The `COMPOSE`-operation is the inverse of the `SPLIT`-operation, which mean that if O is an object of concept `SPATIAL` then O=COMPOSE(SPLIT(O)) holds.
- Finally, the `APPROX` operation approximates a spatial object or a set of spatial objects with a new spatial object. This new spatial object is a representative of the data space covered by the objects to be approximated. The typical and most common approximation is the n-dimensional bounding rectangle. However, the approx operation is not restricted to the bounding rectangle — arbitrary operations can be used as long as they satisfy the condition of dominating predicates [23]. In our context this means that e.g. if the approximations of two objects do not overlap, the two original object must not overlap.

CONCEPT `SPATIAL`		
`OVERLAPS`	spatial_object1, spatial_object2	→ `INT`
`SPLIT`	spatial_object	→ { spatial_object }
`COMPOSE`	{ spatial_object }	→ spatial_object
`APPROX`	{ spatial_object }	→ spatial_object

Table 1. The four concept typical operations of the `SPATIAL` concept.

Table 1 summarizes these four concept typical operation of the `SPATIAL` concept. The user implements these four operations for his application, query and index objects and registers them to the Concert kernel system.

In addition to the concept typical operations, a few basic object management operations such as a copy operation are needed. Using the management operations and the concept typical operations of the four concepts, Concert implements generic methods of physical database design and can deal each data type belonging to one of the concepts for which the necessary management and concept typical operations are implemented, independent of the concrete implementation of the data types. Indexing and query processing in the DBMS kernel is performed based only on these operations. For more information about the concepts, the operations and the CONCERT system in general see [14, 18, 17, 2].

In the next section we use this basic approach to data management to construct a generic spatio-temporal index based on a R-tree like structure which can be used as a framework for spatial indexing.

3 A Framework for Generic Spatio-Temporal Data Indexing

In order to keep the explanation of the generic spatio-temporal index and its integration into the CONCERT kernel system simple and understandable, we explain, how the well-known R-tree [7] could be implemented and generalized in this context. The R-tree certainly may not be optimal for indexing spatio-temporal objects, especially for temporal indexes (see [22] for more information), however, due to its simple and well-understood structure, it is useful to explain the extensibility aspects. We will abstract the R-tree approach to deal with concepts. In this way we get an index structure which is

- generic in the sense of data being used, because it is based only on the "behaviour" of it (the concept it belongs to) and
- generic in the sense of algorithms, because we use a generic heuristic functions which determine the tree-internal processes (varying the heuristics can change the R-tree like behaviour into an R+-tree or an other derivation), and
- generic in the sense of nodes covering search spaces, because we use spatio objects to approximate the space of subtrees and no fixed spatial shapes and which
- uses the features of the underlying CONCERT kernel system like abstract data type, arbitrary object size and management of distributed and external repositories.

3.1 From R-trees to Generic Trees

The traditional R-tree basically works as described in the following: It is a height-balanced tree with nodes of equal size storing pointers to data objects in its leaf nodes. Each data object to be indexed in the R-tree is represented by a minimal bounding rectangle. Associated with each leaf node is a rectangle defined as the minimal bounding rectangle of all object rectangles contained in the node. Inner nodes contain pointers to subtrees and have a rectangle associated defined as the minimal bounding rectangle of all objects in the subtree. Therefore the R-tree hierarchy corresponds to hierarchies of rectangular subspaces.

Bounding Boxes vs. SPATIAL Objects: We generalize the R-tree in the following aspects: While the R-tree is restricted to store rectangular data objects, we allow any objects conforming to the `SPATIAL` concept to be stored in the generic spatio-temporal tree. We exploit the Concert low-level storage capabilities providing an efficient multi-page storage system. Therefore, we do not restrict the generic spatio-temporal tree to have fixed node size. Nodes can be enlarged dynamically to virtually unlimited[4] size using a multi-page secondary storage allocation scheme [1]. The R-tree nodes have minimal bounding rectangles associated with them, while the generic spatio-temporal tree uses abstract spatial objects instead (e.g. spheres, convex polygons or just rectangles). These spatial objects are usually computed using the `APPROX` operation.

The only assumption we make here, which is implicit given by the `SPATIAL`-concept, is the existence of an `APPROX`-operation evaluable on node objects.

Object Lookup: The usual way of accessing spatial data in an R-tree is to start with a rectangular query window. At each level of the tree, all subtrees with non-empty overlap of the query window and the bounding rectangle of the subtree are recursively searched for matching objects.

The generic spatio-temporal tree uses a `SPATIAL` object to describe the data space of the query window, or more generally, the arbitrary query space.

With this knowledge the R-Tree algorithms as given in [7] can be extended to a a generic algorithm using the `SPATIAL` concept and its concept typical operation `OVERLAP`, as shown in Algorithm 1. Such algorithms can easily incorporated into the Concert system.

```
find (query, node)
for all e ∈ { node.entries } do
   if leafnode(node) then
      if OVERLAPS (query, e.object) then
         report e.object
      end if
   else
      if OVERLAPS (query, e.region) then
         find (query, e)
      end if
   end if
end for
```

Algorithm 1: Lookup in a generic spatial index.

The concept typical operation used for spatial index lookup is the `OVERLAPS` operation. Note that the Concert spatial index is much more flexible than any given tree variant. If the abstract objects stored in the nodes are minimum bounding rectangles, and query objects are rectangles as well, Algorithm 1 behaves exactly as the R-tree lookup. Because Concert makes almost no assumption about the objects in the tree, Algorithm 1 works the same way also

[4] There is a hard limit of 4 GByte per node. However, in order to be efficient, inner nodes should not become larger than about 1 MByte.

for arbitrary n-dimensional spatial objects, as long as the subtrees form hierarchies of data spaces. Certain applications might prefer to e.g. use overlapping convex polygons to partition the data space or a sphere, if Euclidean distance is important in queries like nearest neighbor queries for point data.

If the objects contain, beside the spatial, a temporal dimension, Algorithm 1 can be used directly for spatio-temporal objects. Note, that it is the responsibility of the user implementing the `OVERLAP` operation to distinguish between the spatial and the temporal dimension.

Splitting a Node - The Dead Space Revival: One of the important issues often discussed for R-trees in the context of spatio-temporal applications is the problem of dead space in the rectangles. The larger the rectangles are compared to the data space covered by the objects contained in the node, the less efficient the R-tree becomes. Our generic tree provides an easy and flexible solution to this problem.

Index nodes as well as data objects stored in the index are spatially extended objects implementing the operations of the `SPATIAL` concept. Therefore, large objects can be split into several (smaller) ones by using the concept typical operation `SPLIT`. This operation can be called multiple times until the resulting objects have — from an application point of view — a good size. It is not the database system and its index structure that determines neither the split granularity nor the split boundaries. If from an application point of view, there is no point in splitting an object further, the `SPLIT` operation just returns the unchanged object.

Note that the `SPLIT` operation is much more powerful than just splitting an rectangle in multiple ones. The `SPLIT` operation needs an object of an arbitrary data type following the `SPATIAL` concept and returns a set of object. The only requirement is, that the resulting objects follow the `SPATIAL` concept. Such objects might be e.g. rectangles as in R-trees or spheres as in M-trees. The exact behaviour of a `SPLIT` operation is determined by its implementation.

As discussed earlier, CONCERT has virtually no size restriction for its index nodes. Using the `OVERLAPS` operation, the spatial index code therefore can handle arbitrary large objects — it just might not be very efficient, if the `SPLIT` operation is not actually splitting the objects. In any case, splitting is done exploiting application semantics rather than following a node-space constraint.

Splitting is possible in different situations. An important one is the a priori splitting of objects at insertion or update time. Such approaches are included in R-tree derived trees. By using the concept typical `SPLIT` operation we generalize these well studied split procedure. The concrete implementation of the operation can determine different application dependent heuristics adapted to the requirements.

One main reason for splitting objects is to avoid dead space in the tree nodes decreasing the efficiency of the index, This is done e.g. in R+-trees. Especially in spatio-temporal applications, objects might change their spatial extension with respect to the time dimension and therefore increase the dead space of their spatio-temporal approximation. Finally, splitting also can occur when performing a node split in the index tree during inserting or in case of reorganization of the tree.

Insertion of Objects: In principle, the insertion of objects into our generic spatial tree index follows the same steps as inserting them into any other tree like indexing structure. We generalize this approach – similar to the GiST approach in [9] – by encapsulating the tree strategies like e.g. "When should a node/object be splitted?" in single methods implementing the best heuristic for an application. In contrast to [9] we develop our algorithm based on the concept typical operations and the tree typical operations adapt to the concept driven approach whereas GiST follows a tree structure driven approach, generalizing operations in the context of tree management. Algorithm 2 shows the generic insertion procedure for our index.

```
insert (object, node)
if leafnode(node) then
   Consider Split
   Insert Object
else
   Choose Subtree
   insert (object, subtree)
   Consider Subtree-Split
end if
Adjust Node
```

Algorithm 2: Insert into a generic spatial index.

Although similar from its outline, the generic spatial index has some important differences to the R-tree. While in the R-tree, nodes are of fixed size and therefore, a node has to be split according to application requirements, the generic index is more flexible. The operation `Consider Split` can implement a flexible splitting heuristics considering not only the size of the node, but also the spatial extension of the objects and the amount of dead space in the node. In this way the concrete choice of a heuristic determines the behaviour of the tree, let him behave like a R-, a R+- or any other indexing tree for spatial data.

The splitting itself is also more flexible. It can be performed not only by distributing the entries among the nodes (using an arbitrary splitting strategy such as one of the strategies discussed in [7]), but also by splitting large objects using the concept typical operation `SPLIT` reducing the dead space further. In this way, the implementation of `SPLIT` controls a part of the behaviour of our indexing framework.

Even the insertion of an object in a subtree is more flexible than in a concrete tree implementation: The insertion is handled by recursively passing the object down the tree. At each non-leaf node, an appropriate subtree has to be chosen for the recursion. This is done by the operation `Choose Subtree`. In the standard R-tree algorithm, the subtree is chosen based on the least enlargement of the bounding rectangle necessary. In the generic spatial index, the concept typical operation `OVERLAPS` is used. In order to optimize the choice for a subtree, `OVERLAPS` is not defined as a simple predicate but rather returns an integer value indicating the amount of overlap. In addition to the amount of overlap, the current size of the subtree and the amount of free space in the subtree can also be considered.

The additional flexibility over fixed multidimensional indexing tree gained with the operations `Choose Subtree`, `Consider Split` and `Consider Subtree--Split` together with the mechanism avoiding dead space using the concept typical operation `SPLIT` makes our approach to a useful framework of an R-tree like index for spatio-temporal applications.

The heuristics used in the algorithms are application and environment dependent and an important topic in ongoing research activities, e.g. in the CHOROCHRONOS[3] project. Some popular heuristics like these in R- or R+-trees are well known but not always the best choice in case of complex spatio-temporal data.

```
adjust node (node)
Consider Reorganization
Adjust covering subspace
if node has been split then
   Create covering subspace of new node
   Propagate Split
end if
```

Algorithm 3: Adjust an index node.

Reorganizing the Tree: After insertion, nodes have to be adjusted using the `Adjust Node` operation as shown in Algorithm 3. Insertion into the generic spatial tree can bring the tree out of balance. This can be avoided by reorganizing the node moving some of its entries to sibling nodes. However, this operation might become very expensive and is therefore usually omitted. Rather from time to time, when the tree gets too much out of balance, a complete reorganization is performed.

Remembering that each node has a covering subspace object associated with it which follows the `SPATIAL` concept (e.g. a bounding rectangle in R-trees), this object has to be adjusted on insertion using the concept typical operation `APPROX` if reorganisation is necessary. Whether a node has to be adjusted is determined by the `Concider Reorganization` operation which implements the desired heuristic.

If the node has been split, the `APPROX` operation has to be calculated for both nodes and the split has to be propagated to the parent node. If the root node is split, the tree grows by one level.

Deletion of Objects: For object deletion, two strategies can be followed. It is possible always to recompute the spatial extent of each node using the `APPROX` concept typical operation. This keeps the space covered by each node minimal, but it requires substantial overhead each time, an object is deleted.

Alternatively, the spatial extent of the nodes is left unchanged. Deletion is more efficient and no deletion or adjustment of inner nodes will be necessary. If a node becomes too small, it can be merged with one of it's siblings. This keeps small the overhead for reorganizing the tree but decreases the efficiency of the index due to possible dead space in the covering objects of an node.

It is important to notice, that throughout the whole code of our generic spatio-temporal index, no explicit assumption is made about the data types and storage formats of the spatio-temporal data. Only the concept typical operations are used as an interface. The objects themselves are simply treated as abstract objects, i.e. uninterpreted byte sequences with a few operations defined on them. Therefore, it is irrelevant to the kernel system, where the real data objects reside — as long as they can be accessed via the concept typical operations. Instead of objects themselves, it is possible to store only place holders (e.g. a URL or any sort of a pointer to the actual object) and access the real objects only when processing the concept typical operations, for example via remote procedure call. This fact allows the kernel system to cope with the interoperability issue. The actual data can reside in heterogeneous, distributed repositories. The kernel only needs to know the operations and handles to access it and provide physical design and query capabilities over the (external) data. With this in mind we presented a framework of a generic tree index in the last section. It is

- generic in the sense of data type – it is valid and useable for all data type following the `SPATIAL` concept of Concert independent from their location or size – and
- generic in the sense of tree behaviour – changing the tree heuristics leads to different tree derivations – and it is
- derived from the main issue of each data management system – the data – in contrast to approaches which derive their genericity by generalizing algorithms or methods.

4 Conclusions

Advanced applications require the DBMS to be extended with additional application-specific functionality. We presented the Concert extensible architecture that provides a flexible base for spatio-temporal and advanced data handling.

We have introduced the CONCERT kernel system, it's mechanism of physical database design for external objects, and a generic indexing framework as an example for how to exploit extensibility in order to better support advanced indexing for query processing of abstract objects.

Using the well-known R-tree as an example, we have shown how such a generic tree index can be implemented and adapted to spatio-temporal application needs. The investigations the generic approach presented here is not complete yet and has to be evaluated and compared with "hard-wired" indexes. The potential for improvements particularly with respect to the heuristics must be explored and is subject of further research activity. We expect the framework presented to be flexible enough and expect acceptable overhead to be traded for increased flexibility.

4.1 Acknowledgement

We thank the reviewers for their detailled comments that have helped us to revise and to focus the paper.

References

1. Stephen Blott, Helmut Kaufmann, Lukas Relly, and Hans-Jörg Schek, *Buffering long externally-defined objects*, Proceedings of the Sixth International Workshop on Persistent Object Systems (POS6) (Tarascon, France) (M.P. Atkinson, V. Benzaken, and D. Maier, eds.), British Computer Society, Springer Verlag Berlin Heidelberg New York, September 1994, pp. 40–53.
2. Stephen Blott, Lukas Relly, and Hans-Jörg Schek, *An open abstract-object storage system*, Proceedings of the 1996 ACM SIGMOD Conference on Management of Data, June 1996.
3. *Chorochronos: A research network for spatiotemporal database systems*, http://www.dbnet.ece.ntua.gr/ choros/.
4. Douglas Comer, *The ubiquitous B-Tree*, ACM Computing Surveys **11** (1979), no. 2, 121–137.
5. James O. Coplien and Douglas C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, ISBN 0-201-60734-4.
6. Volker Gaede and Oliver Günther, *Multidimensional access methods*, ACM Computing Surveys 30 (3), 170–231 (Sept. 1, 1998).
7. Antonin Guttman, *R-Trees: A dynamic index structure for spatial searching*, ACM SIGMOD International Conference on Management of Data (1984), 47–57.
8. L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, *Starburst mid-flight: As the dust clears*, IEEE Transactions on Knowledge and Data Engineering **2** (1990), no. 1, 143–160.
9. Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer, *Generalized search trees for database systems*, Proceedings of the 21st International Conference on Very Large Databases (Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, eds.), September 1995, pp. 562–573.
10. *IBM corporation*, http://eyp.stllab.ibm.com/t3/.

11. *IBM* white paper, *Building Object-Oriented Frameworks*, http://www.ibm.com/-java/education/oobuilding/buildingoo.pdf.
12. *Informix corporation*, http://www.informix.com/informix/products/options/-udo/datablade/dbmodule/index.html.
13. *Oracle corporation*, http://www.oracle.com/st/cartridges/.
14. Lukas Relly, *Offene Speichersysteme: Physischer Datenbankentwurf für externe objekte*, Ph.D. thesis, Eidgenössisch Technische Hochschule (ETH), Zürich, CH-8092 Zürich, Switzerland, 1999.
15. Lukas Relly and Uwe Röhm, *Plug and play: Interoperability in CONCERT*, The 2nd International Conference on Interoperating Geographic Information Systems (Interop99), Lecture Notes in Computer Science, Springer Verlag Berlin Heidelberg New York, March 1999.
16. L. Rowe and M. Stonebraker, *The POSTGRES data model*, Proceedings of the Thirteenth International Conference on Very Large Databases (Brighton, England), 1987.
17. Lukas Relly, Hans-J. Schek, Olof Henricsson, and Stephan Nebiker, *Physical database design for raster images in Concert*, 5th International Symposium on Spatial Databases (SSD'97) (Berlin, Germany), July 1997.
18. Lukas Relly, Heiko Schuldt, and Hans-J. Schek, *Exporting database functionality — the Concert way*, Data Engineering Bulletin **21** (1998), no. 3, 43–51.
19. Hans-Jörg Schek, H.-B. Paul, M.H. Scholl, and G. Weikum, *The DASDBS project: Objectives, experiences, and future prospects*, IEEE Transactions on Knowledge and Data Engineering **2** (1990), no. 1, 25–43.
20. M. Stonebraker, L. Rowe, and M. Hirohama, *The implementation of POSTGRES*, IEEE Transactions on Knowledge and Data Engineering **2** (1990), no. 1.
21. Michael Stonebraker, *Inclusion of new types in relational database systems*, Proceedings of the International Conference on Data Engineering (Los Angeles, CA), IEEE Computer Society Press, February 1986, pp. 262–269.
22. Betty Salzberg and Vassilis J. Tsotras, *A comparison of access methods for temporal data*, TimeCenter Technical Report **TR-18** (1997).
23. P.F. Wilms, P.M. Schwarz, H.-J. Schek, and L.M. Haas, *Incorporating data types in an extensible database architecture*, Proceedings of the 3rd International Conference on Data and Knowledge Bases, Jerusalem, June 1988.