LNCS 2520, pp263-318, October 2003.
(URL: http://www.springerlink.com/link.asp?id=5g30rx5rx20e71my)

7 Architectures and Implementations of Spatio-temporal Database Management Systems

Martin Breunig[1], Can Türker[2], Michael H. Böhlen[5], Stefan Dieker[4], Ralf Hartmut Güting[4], Christian S. Jensen[5], Lukas Relly[3], Philippe Rigaux[6], Hans-Jörg Schek[2], and Michel Scholl[6]

[1] Uni Vechta, Germany
[2] ETH Zürich, Switzerland
[3] UBS AG Zürich, Switzerland
[4] FernUniversität Hagen, Germany
[5] Aalborg University, Denmark
[6] CNAM, Paris, France

7.1 Introduction

This chapter is devoted to architectural and implementation aspects of spatio-temporal database management systems. It starts with a general introduction into architectures and commercial approaches to extending databases by spatio-temporal features. Thereafter, the prototype systems Concert, Secondo, Dedale, Tiger, and GeoToolKit are presented. As we will see, the focus of these systems is on different concepts and implementation aspects of spatial, temporal, and spatio-temporal databases, e.g. generic indexing, design of spatial, temporal, and spatio-temporal data types and operations, constraint modeling, temporal database management, and 3D/4D database support. A comparison of the prototype systems and a brief résumé conclude the chapter.

7.2 Architectural Aspects

To support spatio-temporal applications, the adequate design of a system architecture for a spatio-temporal database management system (STDBMS) is crucial. Spatio-temporal applications have many special requirements. They deal with complex objects, for example objects with complex boundaries such as clouds and moving points through the 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 sources.

To build an STDBMS, the traditional DBMS architecture and functionality have to be extended. Managing spatio-temporal data requires providing spatio-temporal data types and operations, extensions to the query and data manipulation language, and index support for spatio-temporal data. Such issues arise not only in a spatio-temporal context but also when building spatial only or temporal only systems. Over the recent years we witnessed three base variants of extending system architectures (see Figure 7.1):

T. Sellis et al. (Eds.): Spatio-temporal Databases, LNCS 2520, pp. 263–318, 2003.

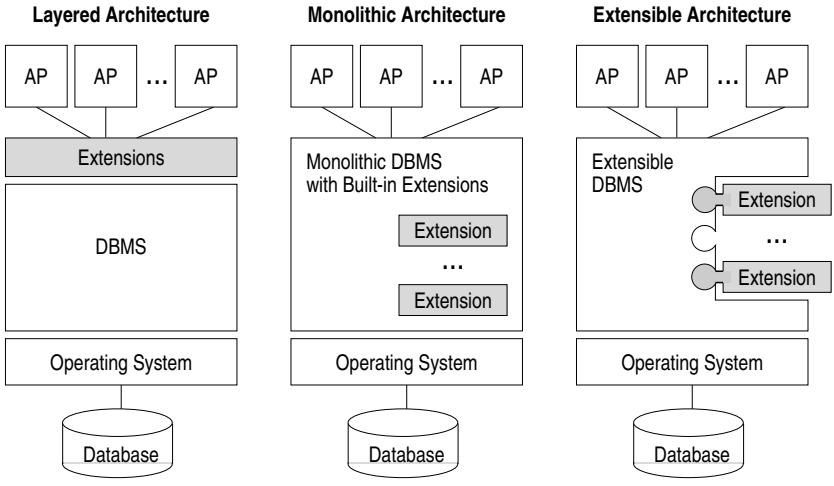

Fig. 7.1. Comparison of system architectures

1. The *layered* approach uses an off-the-shelf database system and extends it by implementing the missing functionality on top of the database system as application programs.
2. In the *monolithic* approach, the database manufacturer integrates all the necessary application-specific extensions into the database system.
3. The *extensible* approach provides a database system which allows to plug user-defined extensions into the database system.

The following subsections present these variants in more detail.

7.2.1 The Layered Architecture

A traditional way of designing an information system for advanced data types and operations is to use an off-the-shelf DBMS and to implement a layer on top providing data types and services for the specific application domain requirements. The DBMS with such a generic component is then used by different applications having similar data type and operation requirements. Such enhanced DBMSs exploit the standard data types and data model — often the relational model — as a basis. They define new data types and possibly a new layer including data definition and query language, query processing and optimization, indexing, and transaction management specialized for the application domain. The new data types are often mapped to low-level data storage containers usually referred as binary large objects of the underlying DBMS. Applications are written against the extended interface. Data definition, queries, and update operations are transformed from the application language to the underlying DBMS interface.

Many of the layered architecture systems have originally been designed as stand-alone applications directly using the operating system as their underlying

storage system. In order to exploit generic database services such as transaction management, their low-level storage system has been replaced by an off-the-shelf database system. Since most of these systems supported only a few standard data types, the layered architecture systems have to use the operating system's file system to store the application-specific data types while using the DBMS to store the standard data types. This architecture is also called the *dual* architecture (see Figure 7.2). Todays advanced DBMSs support binary large objects, and thus get rid of the need to store a part of the data directly in the operating system bypassing the DBMS.

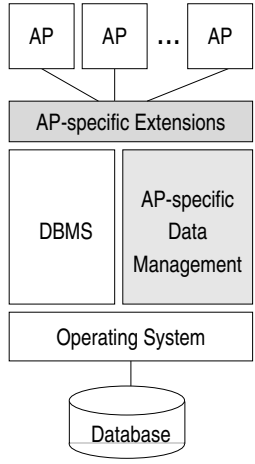

Fig. 7.2. The dual system architecture

The layered approach has the advantage of using standard off-the-shelf components reusing generic data management code. There is a clear separation of responsibilities: application-specific development can be performed and supported independently of the DBMS development. Improvements in one component are directly available in the whole system with almost no additional effort. On the other hand, the flexibility is limited. Development not foreseen in the standard component has to be implemented bypassing the DBMS. The more effort is put into such an application-specific data management extension, the more difficult it gets to change the system and to take advantage of DBMS improvements.

Transaction management is only provided for the standard data types handled by the DBMS. Transactional guarantees for advanced data types as well as application-specific transaction management have to be provided by the application-specific extension. Query processing and optimization has to be performed on two independent levels. Standard query processing and optimization can be handled by the DBMS while spatio-temporal query processing has to take place in the extension. Because system-internal statistics and optimization information are only available inside the DBMS, global query processing and optimization of combined queries is hard to implement. Indexing of standard data types takes

place inside the DBMS while indexing spatio-temporal data has to be dealt with in the extension. Therefore, combined index processing cannot be used.

7.2.2 The Monolithic Architecture

As in the layered architecture, many systems using a monolithic architecture have originally been designed as stand-alone applications. In contrast to the layered architecture, the monolithic architecture extends an application system with DBMS functionality instead of porting it to a standard DBMS. In this way, specialized DBMSs with query functionality, transaction management, and multi-user capabilities are created. The data management aspects traditionally associated with DBMS and the application-specific functionality are integrated into one component.

Because of the tight integration of the general data management aspects and the application-specific functionality, monolithic systems can be optimized for the specific application domain. This generally results in good performance. Standard and specialized index structures can be combined for best results. Transaction management can be provided in a uniform way for standard as well as for advanced data types. However, implementing an integrated system becomes increasingly difficult, the more aspects of an ever-growing application domain have to be taken into account. It might be possible — even though it has proven difficult — to build a monolithic system for a spatial only or a temporal only database. Combining spatial and temporal data management adds another dimension of complexity such that is very difficult to provide a satisfactory solution using the monolithic approach.

7.2.3 The Extensible Architecture

The layered as well as the monolithic architecture do not support an easy adaptation of the DBMS to new requirements of advanced applications. The user, however, should be able to "taylor" the DBMS flexibly according to his specific requirements. Extensible database systems provide a generic system capable of being extended internally by application-specific modules. New data types and functionality required for specialized applications is integrated as close as possible into the DBMS. Traditional DBMS functionality like indexing, query optimization, and transaction management is supported for user-defined data types and functions in a seamless fashion. In this way, an extensible architecture takes the advantage of the monolithic architecture while avoiding its deficiencies. It thus provides the basis for an easy integration of advanced spatio-temporal data types, operations, and access methods which can be used by the DBMS analogously to its standard data types and access methods.

The first extensible system prototypes have been developed to support especially non-standard DBMS applications like geographical or engineering information systems. Research on extensible systems has been carried out in several projects, e.g. Ingres [57], DASDBS [69,68], STARBURST [40], POSTGRES [74,75], OMS [10], Gral [35], Volcano [29,30], and PREDATOR [70]. These

projects addressed, among other, data model extensions, storage and indexing of complex objects as well as transaction management and query optimization in the presence of complex objects.

Another way to provide extensible database systems are database toolkits. Toolkits do not prescribe any data model, but rather identify and implement a common subset of functionality which all database systems for whatever data model must provide, e.g., transaction management, concurrency control, recovery, and query optimization. Key projects of that category are GENESIS [5], EXODUS [18], and SHORE [19].

While toolkits are very generic and can be used for the implementation of many different data models, they leave too much expenditure at the implementor to be accepted as a suitable means for fast system implementation. Extensible systems, on the other hand, pre-implement as much functionality as possible at the expense of flexibility, since they usually prescribe a specific data model, e.g., an object-relational model.

The SQL99 standard [48] specifies new data types and type constructors in order to better support advanced applications. As we will see in the next section, commercially leading database vendors already support the development of generic extensions — even though not in its full beauty. These extensible relational systems are referred as *object-relational* systems.

7.2.4 Commercial Approaches to Spatial-temporal Extensions

In the following, we briefly sketch the commercial approaches of Informix, Oracle, and IBM DB2, which exploit extensible, object-relational database technology to provide spatio-temporal extensions. Also, we briefly sketch commercial approaches in the field of geographic information systems.

Informix Datablades. The Informix Dynamic Server can by extended by *datablades* [47]. Datablades are modules containing data type extensions and routines specifying the behavior of the data and extending the query language. New data types can be implemented using built-in data types, collection types like `SET`, `LIST`, or `MULTISET`, or large unstructured data types like `BLOB`. User-defined functions encapsulated in the datablades determine the behavior and functionality of new data types.

A new access method must create an *operator class* containing both a set of strategy functions and a set of support functions. The strategy functions are used for decisions of the optimizer to build execution plans. The support functions are needed to build, maintain, and access the index. Access methods are used by the database server every time an operator in the filter of a query matches one of the strategy functions and results in lower execution costs than other access methods.

The datablade development requires expendable implementation work and a deep understanding of the internals of the system. Therefore, the creation of new access methods and their integration into the optimizer is mostly done

by companies offering commercial products like the text indexing and retrieval datablade of Excalibur [43].

The Informix *Geodetic* datablade [44] extends the server with spatio-temporal data types like `GeoPoint`, `GeoLineseg`, `GeoPolygon`, `GeoEllipse`, `GeoCircle`, and `GeoBox`, and associated operators like `Intersect`, `Beyond`, `Inside`, `Outside`, and `Within`. An R-tree [39] is provided to index spatio-temporal data. A specialized operation class, called `GeoObject_ops`, is available, which associates several operators to the R-tree access method. An R-tree index for a spatio-temporal attribute is defined as follows:

```
CREATE INDEX index_name
ON table_name(column_name GeoObject_ops)
USING RTREE;
```

Informix also provides the *Spatial* datablade [45] as well as the *TimeSeries* datablade [46] to extend the server with data types and functions referring to spatial data only and to temporal data only, respectively.

Oracle Cartridges. The Oracle8 server can be extended by *cartridges* [60]. These modules add new data types and functionality to the database server. As in Informix with datablades, data types and functions can easily be implemented within a cartridge but the integration of indexes into the query optimizer needs deep knowledge of the database server. In addition, access method extensions and their integration into the query optimizer are limited to a small amount of simple query constructions.

Based on the built-in data types, array type, reference type, and large object types, new data types can be implemented within a cartridge. The complexity of data types is limited and not all data types can be combined orthogonally. An index structure is defined within the cartridge. In contrast to Informix, index maintenance – insertion, deletion, and update – must be implemented within the cartridge; no genericity is provided here. The usage of new index structures can be plugged into the optimizer by user-defined functions returning information about execution costs, statistics information, and selectivity of the index for a given predicate. This allows the cost-based optimizer to use new indexes and to build alternative execution plans.

To deal with spatial data, the *Oracle Spatial* cartridge [58] is available. This cartridge extends the database server with data types representing two-dimensional points, lines, or polygons. Based on these primitive data types, composite geometric data types like point cluster and polygons with holes are provided. A linear quad-tree [26] is provided for spatial indexing. A spatial index on a column of type `MDSYS.SDO_GEOMETRY` is defined as follows:

```
CREATE INDEX index_name
ON table_name(column_name)
INDEXTYPE IS MDSYS.SPATIAL_INDEX;
```

This index is then exploited when executing geometry-specific operations like `RELATE`, `SDO_INTERSECTION`, `SDO_Difference`, and `SDO_UNION`.

The *Oracle Time Series* cartridge [59] is provided for temporal data. This cartridge extends the server with calendar and time series data types.

IBM DB2 Extenders. IBM's modules which extend the DB2 server with abstract data types are called *extenders*. Extenders can be implemented using built-in data types, reference types, and large object types.

Spatial data is supported by IBM's *DB2 Spatial* extender. This module extends the DB2 server with spatial data types like `Point`, `Linestring`, and `Polygon` and a rich repertoire of operations like `Contains`, `Intersects`, `Overlap`, `Within`, `Disjoint`, `Touch`, and `Cross`. A grid file [55] is used as spatial index. Such a spatial index is defined as follows:

```
CREATE INDEX index_name
ON table_name(column_name)
USING spatial_index;
```

The database server is then aware of the existence of this index and the query optimizer takes the advantage of the index when accessing and manipulating spatial data. In addition to specifying the costs of a user-defined function, the user can indicate whether or not the function can be taken as predicate in the where clause.

A DB2 *Time Series* extender, which provides similar functionality as its Informix and Oracle counterparts, is offered by a Swedish company [25].

Geographical Information Systems. Geographical Information Systems (GIS) have also been designed according to the different architecture variants discussed in the previous subsections. ARC/INFO [2], for example, is a representative of the dual architecture, whereas ARC/INFO's spatial database engine and SMALLWORLD GIS [72] have a monolithic system architecture. However, hitherto there are no commercial GIS that allow an extension of their database kernel.

As GIS have originally been developed to support the construction of digital maps, they have made much more progress in handling spatial data than temporal data. The key issue with GIS during the last years has been the combined handling of spatial and non-spatial data [54]. Database support for dynamic maps, however, is still a research topic. To the best of our knowledge, there is no efficient support for the handling and processing of temporal data in today's commercial GIS. These systems have not been designed for temporal data support and currently are not easily extensible for temporal data management. A prototype module for the management of temporal data on the basis of a commercial GIS has been proposed by [50].

7.3 The Concert Prototype System

7.3.1 Introduction

CONCERT [9,65,64] is a database kernel system based on an extensible architecture. However, CONCERT propagates a new paradigm called "exporting database functionality" as a departure from traditional thinking in the field of databases. Traditionally, all data is loaded into and owned by the database in a format determined by the database, whereas according to the new paradigm data is viewed as abstract objects with only a few central object methods known to the database system. The data may even reside outside the database in external repositories or archives. Nevertheless, database functionality, such as query processing and indexing, is provided. The new approach is extremely useful for spatio-temporal environments, as there is no single most useful data format to represent spatio-temporal data. Rather, many different spatio-temporal data formats coexist. The new paradigm presented in CONCERT allows them to be viewed in a uniform manner.

While traditional indexing is based on data types — an integer data type can be indexed by a B-tree, a polygon data type can be indexed by an R-tree — CONCERT's indexing is based on the conceptual behavior of the data to be indexed: data that is associated with an ordering can be indexed by a B-tree. Data implementing spatial properties (such as overlaps and covers predicates) can be indexed by an R-tree. CONCERT uses four classes of behavior, `SCALAR`, `RECORD`, `LIST`, and `SPATIAL`, covering all relevant concepts that are the basis of today's indexing techniques.

With these classes conceptual behavior of data is described in contrast to type dependent properties like in GiST [41]. On top of the CONCERT classes, arbitrary indexing structures can be build which are valid for a variety of data types. New indexing structures are implemented for concepts and use concept typical operations. In this way, indexing techniques of the CONCERT approach are independent of the concrete implementation of the data types — in particular it is not limited to tree structures alone. This is the major difference between GiST and the indexing framework provided by CONCERT.

7.3.2 Architecture

Following the extensible system architecture, CONCERT 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. The object manager combines different collections adding join capability to the kernel collections. Figure 7.3 shows an overview of the CONCERT architecture.

The CONCERT kernel system consists of two components, the Storage Object Manager (SOM) and the Abstract Object Manager (AOM). The SOM provides DBMS base services such as segment management, buffer management,

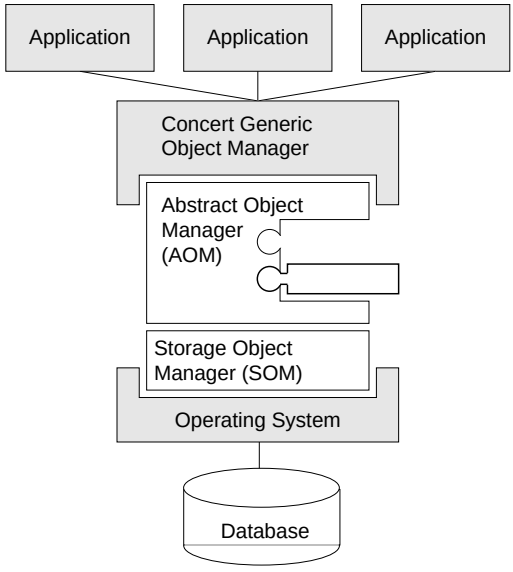

Fig. 7.3. CONCERT system architecture

and management of objects on database pages. It is tightly integrated into the underlying operating system exploiting multi-threading and multi-processing. It uses a hardware-supported buffer management exploiting the operating system's virtual memory management by implementing a memory-mapped buffer [8]. Details of the CONCERT SOM can be found in [63].

The AOM 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 7.4 shows the (simplified) collection interface definition.

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 be built easily. This allows, for example, an

```
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. 7.4. (Simplified) AOM collection interface

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 far 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 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. A small set of concepts is identified to be sufficient to allow most physical design decisions for spatio-temporal DBMS. The next subsection introduces an R-tree like generic index framework with minimal restrictions to the flexibility of external objects.

7.3.3 Spatio-temporal Extensions

To explain CONCERT's database extensibility idea, let us look at a standard B-tree index [21], 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 lexicographically ordered. Another ordering operation is used for `DATE` data types. Much more data types are possible but the central aspect when data is indexed by a B-tree is that there is an attribute which can be ordered [73]. On the other hand, data types which cannot be ordered cannot 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 about the type of the data is not needed to manage it but its conceptual behavior and the *concept typical operations* associated with these concepts must be known. These operations have to be provided by the user to allow CONCERT to interpret a given abstract object storage type. Four concepts 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, for instance, 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 an ordered concatenation of components which themselves are arbitrary concepts. The concept typical operation is the `SUB_OBJECT` operation returning a component of the `RECORD`. A `RECORD` implements 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 or time space alone. The concept typical operations `OVERLAPS`, `SPLIT`, `COMPOSE`, and `APPROX` are explained in the following.

The user has to implement extensions by plugging new data types into the existing framework. Spatio-temporal indexing structures can be implemented and plugged into Concert independent of the data type indexed by just using the `SPATIAL` concept typical operations.

In traditional systems, the class of space covering objects is predefined by the index method provided and thus "hard-wired" in the system, e.g., a space covering rectangle in the R-tree. Therefore, the classes of application objects and query objects also have to be predefined by the DBMS requiring the user to convert his data into the DBMS format. This clearly is not desirable in the context of extensible systems. Concert addresses this problem. 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[1] 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. Therefore it is not defined as a simple predicate operation, but rather as an operation returning

[1] Note that the spatial concept is not restricted to objects of the geometric space. The spatial conceptcan also be used to model object of the temporal space as well as other data spaces whose objects exhibit similar behavior as geometric objects, e.g., supports inclusion and overlapping predicates as well as split and compose operations.

an integer value indicating the degree of overlap. Negative values are used as a measure for indicating the distance between two objects[2]. 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 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 split" 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 buffering is easier. In the second case, the data space in index nodes is smaller allowing a better pruning in the index structure. The behavior depends on the implementation of the `SPLIT` operation.
- The `COMPOSE` operation recombines spatial objects that have previously been split to an "overall" spatial object. This operation is used for the reconstruction of large data objects which have been split 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. We only consider the conceptual behavior of the data. The `COMPOSE` operation is the inverse of the `SPLIT` operation, which means 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 [77]. In our context, this means that, for example, if the approximations of two objects do not overlap, the two original object must not overlap. Figure 7.5 summarizes these four concept typical operations of the `SPATIAL` concept.

The user implements these four operations for an application, query and index objects and registers them to the CONCERT kernel system.

[2] 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.

CONCEPT SPATIAL		
Operation	**Parameter**	**Result**
OVERLAPS	spatial_object1, spatial_object2	SCALAR
SPLIT	spatial_object	{ spatial_object }
COMPOSE	{ spatial_object }	spatial_object
APPROX	{ spatial_object }	spatial_object

Fig. 7.5. Typical operations of the SPATIAL concept

7.3.4 Implementation Details

CONCERT can deal with each data type belonging to one of the concepts for which the necessary management and concept typical operations are implemented, independently of the concrete implementation of the data types. Indexing and query processing in the DBMS kernel is performed based on these operations only. For more information about the concepts, the operations, and the CONCERT system in general see [63,66,65,9]. In the following, we describe how spatio-temporal indexes can be plugged into CONCERT by using a framework for generic spatio-temporal data indexing.

Although the well-known R-tree [39] may not be optimal for indexing spatio-temporal objects (particularly for temporal indexes; see [67] for more information), its simple and well-understood structure is useful to explain the extensibility aspects. The R-tree is implemented and generalized within the CONCERT framework. Abstracting from the R-tree approach results in an index structure which is generic in the sense of data because it is based only on the "behavior" of the concept it belongs to. It is also generic in the sense of algorithms because of the use of generic heuristic functions which determine the tree-internal processes (varying the heuristics can change the R-tree like behavior into an R+-tree or another derivation). Finally, it is also generic in the sense of nodes covering search spaces because of the use of spatial objects to approximate the space of subtrees and no fixed spatial shapes.

Whereas the R-tree is restricted to store rectangular data objects, the CONCERT approach allows any objects conforming to the SPATIAL concept to be stored in the *generic spatio-temporal tree*. The CONCERT low-level storage capabilities provides an efficient multi-page storage system. Therefore, the generic spatio-temporal tree does not need to have a fixed node size. Nodes can be enlarged dynamically to virtually unlimited[3] size using a multi-page secondary storage allocation scheme [8]. The R-tree nodes have minimal bounding rectangles associated with them, whereas 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 made here, which is implicit given by the SPATIAL concept, is the existence of an APPROX operation which can be evaluated on node objects.

[3] 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.

7.3.5 Case Studies

In the following we describe how a generic spatio-temporal tree can be used in CONCERT with its typical operations of object-lookup, insertion with splitting of a node and deletion including the reorganization of the tree.

Object Lookup. 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 [39] can be extended to a generic algorithm using the `SPATIAL` concept and its concept typical operation `OVERLAPS`, as shown in Figure 7.6. Such algorithms can be incorporated easily into the CONCERT system.

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

Fig. 7.6. Lookup in a generic spatial index

The concept typical operation 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, the algorithm depicted in Figure 7.6 behaves exactly as the R-tree lookup. Since CONCERT makes almost no assumption about the objects in the tree, the algorithm works the same way also for arbitrary n-dimensional spatial objects, as long as the subtrees form hierarchies of data spaces. Certain applications might prefer to use overlapping convex polygons to partition the data space or a sphere, if the Euclidean distance is important like in nearest neighbor queries for point data.

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

Splitting a Node. 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 with respect 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 many times until the resulting objects have a good size from an application point of view. It is not the database system and its index structure that determines the split granularity or 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 a rectangle into many others. The `SPLIT` operation needs an object of an arbitrary data type following the `SPATIAL` concept and returns a set of objects. The only requirement is that the resulting objects follow the `SPATIAL` concept. Such objects might be, for instance, rectangles as in R-trees or spheres as in M-trees. The exact behavior 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 can therefore 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 by exploiting application semantics rather than following a node-space constraint.

Splitting is possible in different situations. One important situation 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 these well-studied split procedures are generalized. The concrete implementation of the operation can determine different application dependent heuristics adapted to the requirements.

Insertion of Objects. In contrast to [41], 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. Figure 7.7 shows the generic insertion procedure for our index.

Although similar to its outline, the generic spatial index has some important differences to the R-tree. Whereas in the R-tree nodes are of fixed size and, hence, 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 behavior of the tree such that it behaves like an R-tree, an R+-tree, 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

```
insert (object, node)
begin
  if leafnode(node)
    Consider Split {Heuristics for splitting or enlarging leaf node}
    Insert Object to leaf node
  else
    Choose Subtree {Heuristics for choice of best subtree}
    insert (object, subtree)
    Consider Subtree Split {Heuristics for splitting or enlarging inner node}
  endif
  Adjust Node
end
```

Fig. 7.7. Insertion into a generic spatial index

such as one of the strategies discussed in [39]) 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 behavior 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 multi-dimensional indexing trees 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 the CONCERT approach a useful framework of an R-tree like index for spatio-temporal applications.

Reorganizing the Tree. After insertion, nodes have to be adjusted using the AdjustNode operation as shown in Figure 7.8. Insertion into the generic spatial tree can bring the tree out of balance. This can be avoided by reorganizing the nodes by moving some of its entries to sibling nodes. However, such a reorganization can be very expensive. Therefore, a complete reorganization is performed at given points in time when the tree gets too much out of balance.
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 operation APPROX if reorganization is necessary. Whether a node has to be adjusted is determined by the *Consider*

```
adjust node (node)
begin
  Consider Reorganization {Heuristics for reorganization with sibling nodes}
  Adjust Covering Subspace {using the APPROX operation}
  if node has been split
    Create Covering Subspace of new node
    Propagate Split to parent node
  endif
end
```

Fig. 7.8. Adjust an index node

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 always possible to recompute the spatial extent of each node using the APPROX 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 since no deletion or adjustment of inner nodes is necessary. If a node becomes too small, it is merged with one of its siblings. This keeps the overhead of reorganizing the tree low, but at the same time decreases the efficiency of the index due to possible dead space in the covering objects of a node.

The CONCERT prototype has been tested, among others, with an application from photogrammetry targeting raster image management [65]. The focus has been on physical database design and query processing for raster image management.

7.4 The Secondo Prototype System

7.4.1 Introduction

SECONDO is a data-model independent environment for the implementation of non-standard database systems. Its goal is to offer the advantages of both the extensible system and the toolkits approach (see Section 7.2.3) while avoiding their disadvantages.

The core component of SECONDO is a generic "database system frame" that can be filled with implementations of a wide range of data models, including, for example, relational, object-oriented, graph-oriented, sequence-oriented, or spatio-temporal database models.[4] The key strategy to achieve this is the separa-

[4] Some of the goals of SECONDO and Concert are similar. A comparison of the two systems is given at the end of Section 7.4.3.

tion of the data model independent components and mechanisms in a DBMS (the *system frame*) from the data-model dependent parts. Nevertheless, the frame and the "contents" have to work together closely. With respect to the different levels of query languages in a DBMS, we have to describe the system frame:

- the *descriptive algebra*, defining a data model and query language,
- the *executable algebra*, specifying a collection of data structures and operations capable of representing the data model and implementing the query language, and
- the *rules* to enable a query optimizer to map descriptive algebra terms to executable algebra terms, also called *query plans* or *evaluation plans*.

A general formalism serving all these purposes has been developed earlier, called *second-order signature (SOS)* [36]. It is reviewed in Section 7.4.2.

On top of the descriptive algebra level there may be some syntactically sugared language, e.g., in an SQL-like style. We assume that the top-level language and the descriptive algebra are entirely equivalent in expressive power; only the former may be more user-friendly whereas the latter is structured according to the SOS formalism. A compiler transforming the top-level language to descriptive algebra can be written relatively easily using compiler generation tools, since it just has to perform a one-to-one mapping to the corresponding data definitions and operations.

At the system level, definitions and implementations of type constructors and operators of the executable algebra are arranged into *algebra modules*, interacting with the system frame through a small number of well-defined support functions for manipulation of types and objects as well as operator invocation. Those algebra support functions dealing with type expressions will be created automatically from the corresponding SOS specification.

7.4.2 Second-Order Signature

Since the SECONDO system implements the framework of second-order signature (SOS) [36], it is necessary to recall the essential concepts here. The basic idea of SOS is to use two coupled signatures to describe a data model as well as an algebra over that data model. To distinguish the two levels of signature, we call the first *type signature* and the second *value signature*. A signature in general has *sorts* and *operators* and defines a set of *terms*.

Specifying a Descriptive Algebra. The type signature has so-called *kinds* as sorts and *type constructors* as operators. The terms of the type signature are called *types*. In the sequel, we show example specifications for the relational model and a relational execution system. Although the purpose of SECONDO is not to reimplement relational systems, it makes no sense to explain an unknown formalism using examples from an unknown data model. The structural part of the relational model can be described by the following signature:

kinds IDENT, DATA, TUPLE, REL
type constructors

	$\rightarrow$	DATA	*int*, *real*, *string*, *bool*
$(\text{IDENT} \times \text{DATA})^+$	$\rightarrow$	TUPLE	*tuple*
TUPLE	$\rightarrow$	REL	*rel*

Here *int*, *real*, *string*, and *bool* are type constructors without arguments, or *constant* type constructors, of a result kind called DATA. A kind stands for the set of types (terms) for which it is the result kind. For DATA this set is finite, namely DATA = {*int*, *real*, *string*, *bool*}. In contrast, there are infinitely many types of kind TUPLE or REL. For example,

tuple([(name, *string*), (age, *int*)])
rel(*tuple*([(name, *string*), (age, *int*)]))

are types of kind TUPLE and REL, respectively. The definition of the *tuple* type constructor uses a few simple extensions of the basic concept of signature that are present in the SOS framework. For example, if $s_1, \ldots, s_n$ are sorts, then $(s_1 \times \cdots \times s_n)$ is also a sort (*product* sort), and if s is a sort, then s^+ is a sort (*list* sort). The term $(t_1, \ldots, t_n)$ belongs to a product sort $(s_1 \times \cdots \times s_n)$ if and only if each t_i is a term of sort s_i; the term $[t_1, \ldots, t_m]$, for $m \geq 1$, is a term of sort s^+ if and only if each t_i is a term of sort s. The kind IDENT is predefined (and treated in a special way in the implementation in SECONDO). Its type constructors are drawn from some infinite domain of "identifiers". Hence they can be used as attribute names here.

The notion of a relation *schema* has been replaced by a relation type, and "relation" is not considered to be a single type, but a type constructor. Hence operations like selection or join are viewed as polymorphic operations. Note that the choice of kinds and type constructors is completely left to the designer of a data model. In contrast to the CONCERTapproach, we are not offering a toolbox with a fixed set of constructors such as *tuple*, *list*, *set*, etc., but instead a framework where new constructors can be defined. Hence the SECONDO system frame as such knows nothing about *rel* or *tuple* constructors, in contrast to CONCERTwhere, for instance, the record concept requires to provide component access.[5] In summary, the terms of the type signature of an SOS specification define a *type system*, which within the descriptive algebra is equivalent to a DBMS data model.

Now the value signature is used to define operations on the types generated by the type signature. Whereas a signature normally has only a small, finite number of sorts, the second level of signature in SOS generally has to deal with infinitely many sorts. Because of this, we write *signature specifications*. The basic tool is *quantification over kinds*. We define a few example operations for the relational model above:

[5] The only predefined kinds and type constructors in SECONDO are IDENT with its "type constructors" and the type constructor *stream* which plays a special role in query evaluation.

operators
∀ *data* in DATA.
data × *data* → *bool* =, ≠, <, ≤, >, ≥

Here, *data* is a type variable ranging over the types in kind DATA. Hence, it can be bound to any of these types which is then substituted in the second line of the specification. So, we obtain comparison operators on two integers, two reals, etc. Relational selection is specified as follows: **pipes**

∀ *rel*: *rel*(*tuple*) in REL.
rel × (*tuple* → *bool*) → *rel* **select**

Here, *rel*(*tuple*) is a *pattern* in the quantification which is used to bind the two type variables *rel* and *tuple* simultaneously. Hence, the first argument to **select** is a relation of some type *rel*, and the second argument is a function from its tuple type to *bool*, that is, a predicate on this tuple type. The result has the same type as the first argument. The second argument of **select** is based on the following extension of the concept of signature defined in SOS: If, for $n \geq 0$, $s_1, \ldots, s_n$ and s are sorts, then $(s_1 \times \cdots \times s_n \rightarrow s)$ is a sort (*function* sort). Furthermore,

$$\mathbf{fun}(x_1 : s_1, \ldots, x_n : s_n)\ t$$

is a term of sort $(s_1 \times \cdots \times s_n \rightarrow s)$ if and only if t is a term of sort s with free variables $x_1, \ldots, x_n$ of sorts $s_1, \ldots, s_n$, respectively.

An operator **attr** allows us to access attribute values in tuples:

∀ *tuple*: *tuple*(*list*) in TUPLE, *attrname* in IDENT, *member*(*attrname*, *attrtype*, *list*).
tuple × *attrname* → *attrtype* **attr**

Here, *member* is a *type predicate* that checks whether a pair (x, y) with $x =$ *attrname* occurs in the *list* making up the tuple type definition. If so, it binds *attrtype* to y. Hence, **attr** is an operation that for a given tuple and attribute name returns a value of the data type associated with that attribute name. Type predicates are implemented "outside" the formalism in a programming language. Precisely the same formalism (although we have not yet seen all of it) can be used to define an *executable algebra*, thereby specifying an execution system for some data model. In this case, type constructors represent data structures and operators represent query processing algorithms implemented in the system.

Commands. In the SOS framework, a *database* is a pair (T, O), where T is a finite set of *named types* and O is a finite set of *named objects*. A named type is a pair, consisting of an identifier and a type of the current (descriptive or executable) algebra. A named object is a pair, consisting of an identifier and a value of some type of the current algebra. SOS defines six basic commands to manipulate a database, regardless of the data model:

type <identifier> = <type expression>
delete type <identifier>

create <identifier> : <type expression>
update <identifier> := <value expression>
delete <identifier>
query <value expression>

A command can be given at the level of descriptive or executable algebra. In the first case, it is subject to optimization before execution; in the second, it is executed directly. In these commands, a *type expression* is a type of the current type signature, possibly containing names of (previously defined) types in the database. A *value expression* is a term of the current value signature, which may also contain constants and names of objects in the database. The **type** command adds a new named type, **delete type** removes an existing type. The **create** command creates a new object of the given type; its value is yet undefined. The **update** command assigns a value resulting from the value expression which must be of the type of the object. The **delete** command removes an object from the database. The **query** command returns a value resulting from the value expression to the user interface or application. Here are some example commands at the executable algebra level:

type city = *tuple*([(name, *string*), (pop, *int*), (country, *string*)])
type city_rel = *srel*(city)
create cities: city_rel
update cities := {enter values into the cities relation, omitted here}
query cities **feed filter** [**fun** (c: city) **attr**(c, pop) > 1000000] **consume**

More details about the SOS framework can be found in [36]. In [6], a descriptive algebra has been defined for GraphDB [37], an object-oriented data model that integrates a treatment of graphs, which shows that the framework is powerful enough to describe complex, advanced data models.

7.4.3 Architecture

Second-order signature is the formal basis for specifying data models and query languages. In this section, we present the SECONDO system frame, providing a clean extensible architecture, implementing all data-model independent functionality for managing SOS type constructors and operators, and supporting persistent object representations. Extending the frame with algebra modules results in a full-fledged DBMS. In addition to the basic commands, SECONDO provides several other commands, e.g., for transaction management, system configuration, administration of multiple databases, and file input and output.

Overview. Figure 7.9 shows a coarse architecture overview of the SECONDO system. We discuss it level-wise from bottom to top. White boxes are part of the fixed *system frame*, which is independent of the currently implemented data model. Gray-shaded boxes represent the extensible part of the SECONDO system. Their contents differ with specific database implementations.

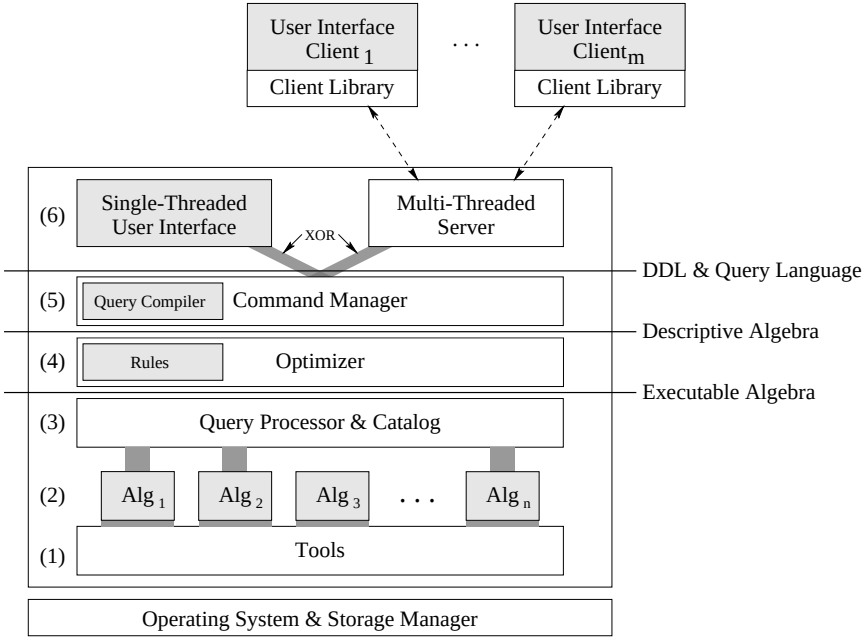

Fig. 7.9. The SECONDO architecture

The system is built on top of the Solaris operating system. Since we want to offer a full-fledged DBMS using a storage manager for dealing with persistent data is essential. In fact, we use the storage manager component of SHORE [19].

At level 1 of the SECONDO architecture, we find a variety of tools, for instance:

- *Nested lists*, a library of functions for easy handling of nested lists, the generic format to pass values as well as type descriptions.
- *SecondoSMI*, a simplified storage manager interface to the SHORE functions used most often. It can be used together with original SHORE function calls whenever the simplified functionality is not sufficient.
- *Catalog tools* for easy creation of system catalogs and algebra-specific catalogs.
- The *Tuple Manager* is an efficient implementation for handling tuples with embedded large objects.
- The *SOS Parser* transforms an SOS term to the generic nested list format used in the system.
- The *SOS specification compiler* creates the source code for the `TypeCheck` and `TransformType` algebra support functions from a valid SOS specification.

Level 2 is the algebra module level. To some extent, an algebra module of SECONDO is similar to ADTs of PREDATOR [71] or Informix datablades [47]. Using the tools of level 1, a SECONDO algebra module defines and implements type constructors and operators of an executable query algebra. SECONDO allows for

implementations in C++, Modula-2, and C. To be able to use a module's types and operators in queries, the module must be registered with the system frame, thereby enabling modules in upper levels to call specific support functions provided by the module. In Figure 7.9, modules 1, 2, and n are *active* since they are connected to the frame, while module 3 is *inactive*. C++ modules are activated by linking them to the system frame. In addition, the activation of C and Modula-2 modules requires insertion of some standardized lines into the body of a predefined startup function.

Level 3 contains the query processor, the system catalog of types and objects (remember that a database is just a set of named types and named objects), and the mechanism for module registration. During query execution, the query processor controls which support functions of active algebra modules are executed at which point of time. Input to the query processor is a query plan, i.e. a term of the executable algebra defined by active algebra modules. A detailed description of query processing techniques in SECONDO, including the powerful stream concept, can be found in [22] and, on a more technical level, in [38].

The query optimizer depicted at level 4 transforms a descriptive query into an efficient evaluation plan for the query processor by means of transformation rules. For each algebra module, the database implementor provides a corresponding set of rules as well as algebra support functions supplying information on estimated query execution costs.

The command manager at level 5 provides a procedural interface to the functionality of the lower levels. Depending on the command level, the query (or other command) is passed either to the query compiler, provided by the database implementor, to the optimizer or to the query processor.

At level 6 we find the front end of a SECONDO installation, providing the user interface. In general, there are two mutually exclusive alternatives: either the user interface is linked with the frame and active algebra modules to a self-contained program, or the SECONDO process is made a server process serving requests of an arbitrary number of client processes which implement the user interfaces. In the first case, SECONDO is a single-user, single-process system, while in the latter case SECONDO is a multi-user capable client-server system, exploiting the multi-threaded environment offered by SHORE. To support the implementation of user clients, SECONDO provides comfortable client libraries for C++ and Java.

Algebra Modules. An algebra module has to implement a set of support functions for all of its type constructors and operators. Figure 7.10 lists all support functions for type constructors implemented in algebra modules. In addition to these functions, also the type constructor name is passed to the system frame. During query processing, the `In` function is used to convert an external value (in nested list form), given as part of an input file or in an interactive query command, to its corresponding internal value. The internal value typically is a catalog index essentially referencing either a main memory pointer or a persistent identifier. The `Out` function is the reverse of the `In` function, producing external values for user interfaces or output files.

`In/Out`	Conversion from external to internal value representation and vice versa.
`Create/Delete`	Allocate/deallocate memory for internal value representation.
`TypeCheck`	Validation of type constructor applications in type expressions.
`InModel/OutModel`	Conversion from nested list to internal model representation and vice versa.
`ValueToModel`	Computes a model for a given value.

Fig. 7.10. Support functions for type constructors

The `Create/Delete` pair of support functions is used to allocate memory for a default value of the given type or to delete an object of that type, respectively. Default value creation is performed by the query processor to reserve space for single stream elements, thereby avoiding multiple allocation and deallocation of memory for stream elements of a common stream, which are never accessed simultaneously. The `Delete` function is called later on to deallocate this default memory space as well as all intermediate results generated while processing a query.

The `TypeCheck` function is called whenever a new database object is created to check whether its type conforms to the signature of the underlying data model.

Furthermore, for each type constructor a *model* may be registered which is a data structure containing summary information about a value of the type. The model is a place to keep statistical information such as expected number of tuples, histograms about attribute value distribution, etc. For maintaining models there are three support functions `InModel`, `OutModel`, and `ValueToModel`, as listed in Figure 7.10.

Figure 7.11 presents all support functions for operators. For each operator, its name and the number of `Evaluate` functions are also passed to the system frame. During query execution, the query processor calls the `TransformType` function for type checking. `Select` is needed for the resolution of overloaded operators. The `Evaluate` function(s) do(es) the "real work" by computing an operator's result value from its argument values.

`TransformType`	Computes the operator's result type from given argument types.
`Select`	Selects the correct evaluation function in case of overloading by means of the actual argument types.
`Evaluate`	Computes the result value from input values. In case of overloading, several evaluation functions exist.
`MapModel`	Computes the result model from argument models.
`MapCost`	Computes the estimated cost of an operator application from argument models and costs.

Fig. 7.11. Support functions for operators

Concerning cost estimation, there are two support functions called `MapModel` and `MapCost`. For an operator application, `MapModel` takes the models of the operator's arguments and returns a model for the result. `MapCost` also takes the models of the operator's arguments as well as the estimated costs for computing the arguments and, based on these, estimates and returns the cost for the operator application. By calling these support functions, the optimizer can estimate properties of intermediate results and the cost of query plans or subplans. Furthermore, there is a startup routine for each algebra module which is used to associate type constructors with the kinds containing them, to perform initializations of global arrays, etc.

The registration mechanism for support functions differs from the implementation language. Registration is most comfortable in C++: For each type constructor, an instance of the predefined class `TypeConstructor` must be defined, passing operator support functions as constructor arguments. The same happens with operators and a predefined class `Operator`. For a complete algebra, an instance of a class derived from the predefined class `Algebra` is defined. The constructor of this class is the startup routine of the modules.

Algebra modules need not only to cooperate with the system frame, but also with other algebra modules. Modules implement certain signatures. At the type level, *kinds* are the functions between different signatures. For instance, each type in kind DATA will be a valid attribute type for the *tuple* type constructor. Thus, a type constructor *polygon* is made a type constructor for attribute types by simply adding *polygon* to the type constructors for DATA.

At the implementation level, the interface between system frame and algebra modules does not impose any specific inter-module interaction conventions on the algebra implementation, but rather the algebra implementor is free to define the protocol for interaction with type constructors and operators of his algebra. For C++ implementations there is a general strategy, based upon the inheritance and virtual method mechanisms provided by C++, which allows one to define generic interfaces between modules in a uniform manner as follows.

The basic observation is that the relationship between kinds and type constructors corresponds to the relationship between base classes and derived classes. For each kind K, the algebra module *alg* requiring an interface to values of types in K defines an abstract base class *k_base*. For the implementation of operators in *alg*, typically some support functions for dealing with values of kind K will be necessary. Just these support functions are defined as abstract virtual methods of *k_base*. Whenever a class *tc* in any algebra module is defined to implement a type constructor in kind K, *tc* must be derived from *k_base*: `class tc : public k_base`. For instance, the base class `Data` corresponding to kind DATA contains a virtual method `Compare` which has to be defined within all attribute data type implementations, thereby enabling the generic implementation of the `sort` operator of the relational algebra module.

SECONDO *versus Concert.* A common goal of SECONDO and Concert is to support an easy and efficient implementation of new application-specific data types.

In particular, both systems provide facilities for managing collections of related types, namely *kinds* in SECONDO and concepts in Concert.

However, the facilities offered differ as follows. SECONDO enables a database implementor to define arbitrary new type constructors, operators, and, through the notion of *kinds*, sets of type constructors for data types with common properties. The system frame does not implement any particular data model, but rather provides a generic query processor and powerful tools to implement type constructors and operators.

Concert *concepts* are similar to SECONDO kinds in two major aspects. First, concepts group sets of data types with common properties, and second, a concept sets up a programming interface, which has to be implemented by those new data types that shall fit the concept. That way, general processing techniques are implementable for groups of data types rather than specific types only. In Concert, this is basically exploited for indexing.

As opposed to SECONDO kinds, Concert concepts are hardwired with the system. A database implementor is neither able to add new concepts nor to provide further implementations of concept-specific index structures and operators. On the other hand, Concert concepts already cover a broad range of requirements of new data models, and hardwired implementations are often slightly more efficient than those registered through a general interface, because the adaptation to details of other modules in the system is easier.

Regarding SECONDO, introducing a new kind naturally requires some effort to implement the type constructors and operators applicable to these kinds. However, with a growing number of algebra module implementations the probability grows that a database implementor finds kinds with their operators and type constructors already implemented that fit pretty well the requirements of new data types.

In summary, Concert offers a higher degree of implemented functionality than SECONDO. On the other hand, SECONDO is more flexible with respect to new data models since it does not hardwire a fixed number of concepts, but is open for arbitrary "concepts". In fact, SECONDO could be used to implement the Concert approach by defining all concepts as kinds and index structures as type constructors applicable to these kinds.

7.4.4 Implementing Spatio-temporal Algebra Modules

Because of its generality, particularly with respect to type constructor definitions, SECONDO is very well suited to the implementation of spatio-temporal data models. In fact, the signatures given in Sections 4.3 and 4.4 (see Chapter 4) defining the abstract and the discrete data model, are genuine second-order signatures. Thus, the `TypeCheck`, `TransformType`, and `Select` support functions can be implemented in a straightforward manner. Later on, they will even be generated automatically by the SOS specification compiler.

Apart from the type level, a database implementor has to provide *operator implementations* as well as efficient *memory representations* of data type

instances. In doing so, he is supported by SECONDO's implementation tools. Implementors of spatial data types often encounter the problem that instances of a given type might vary in size heavily. For example, a *polygon* value may consist of only three vertices as well as many thousands of them. SECONDO's *tuple manager* (see Section 7.4.3) can be used to handle tuples with embedded attribute values of varying size in an efficient and comfortable way. For that purpose it offers a new large object abstraction called *FLOB* (Faked Large OBject). FLOBs are implemented on top of the large object abstraction of the underlying storage manager. Depending on a FLOB value's size and access probability, it is either stored as a large object or embedded within the tuple [23].

Allowing for type constructors arbitrarily nesting other types, however, requires implementation techniques beyond those of basic FLOBs. Consider a data type that arises from the application of a type constructor to some other type, e.g., *list*(*polygon*). The representation of such a data type needs to organize a set of representations of values of the argument types. If the argument types employ FLOBs themselves, we immediately arrive at a tree of storage blocks which may be small or large, i.e., a *FLOB tree*. The generalized problem is then to determine an efficient storage layout or *clustering* of the tree, i.e., a partitioning into components stored as large objects. Thus, we extended the basic FLOB functionality, offering *nestable FLOBs*, so supporting a direct and elegant implementation of type constructors [24].

Due to the aspired generality of our clustering tool, we cannot exploit specific schema information to decide on a good clustering. Instead, we consider a FLOB's *size* and *access probability*, as well as the general *FLOB access speed* parameters of the underlying storage manager and operating system. In [24], we deduced a *rank function* $R(C, f)$ that for a given cluster C and a FLOB f not yet assigned to any cluster returns a measure indicating whether it is more efficient to insert f into C or to store it in another large object.

Our tool uses the function R in an algorithm that finds a clustering of a FLOB tree in time linear in the number of FLOBs as follows. The root FLOB starts a new cluster. Then, each son of the root either is included into the root's cluster or starts a new cluster, according to the return value of R. This is done recursively for the entire tree. Even though this algorithm cannot guarantee to return an optimal solution, in all tests performed in an experimental comparison with other algorithms it returned high-quality results.

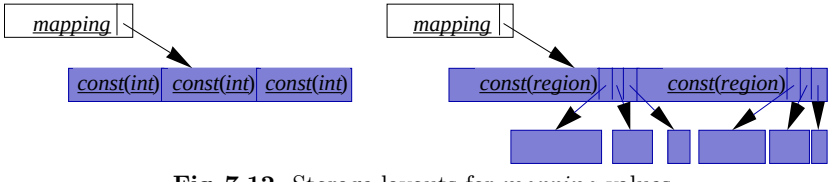

Fig. 7.12. Storage layouts for *mapping* values

As an application example, consider the type constructor *mapping* presented in Section 4.4. The implementation of this type constructor basically provides a persistent *dynamic array* of *unit* values. As the size of the array differs with varying instances of *mapping*, it is implemented as a FLOB. Now the level of nesting depends on the argument types of the mapping, as shown in Figure 7.12. In the case of *const*(*int*) values, there is only one level of nesting. However, in the case of *const*(*region*) values, which are in turn implemented using FLOBs for the different polygons defining a region, another nesting level arises. For database implementors, all shaded boxes in Figure 7.12 are distinct large objects. The much more efficient physical clustering set up by the SECONDO tuple manager is completely hidden to them, so that their focus of attention is left on data type functionality issues.

In summary, SECONDO with its concept of extensibility by algebra modules, fits very well the designs of spatio-temporal algebras presented in Chapter 4 and is an excellent environment for implementing them. Beyond extensibility, it offers powerful tools for managing large objects occurring in such designs, and even supports type constructors organizing collections of such large objects.

7.5 The Dedale Prototype System

7.5.1 Introduction

The DEDALE prototype [32,31] follows the *extensible approach*. It is one of the first implementations of a database system based on the linear constraint model (see Chapter 5). It is intended to demonstrate the relevance of this model to handle geometric applications in areas such as spatial or spatio-temporal applications. One of the interesting features of DEDALE is its ability to represent and manipulate multi-dimensional point-sets as linear constraint relations. [32,31] gave examples of spatio-temporal applications of DEDALE. However, the major contribution of DEDALE to spatio-temporal databases lies on its recent extension to handle *interpolated* spatial data [33]. This model captures the class of geometric objects embedded in a d-dimensional space such that one of the attributes can be defined as a function of a subset of the other attributes. Moreover, this function can be obtained as a *linear* interpolation based upon some finite set of sample values.

DEDALEcovers, among other applications, *moving objects*. Indeed, a trajectory can be represented by a sample of points with time and position. If we make the reasonable assumption that the speed is constant on each segment, the full trajectory can then be recovered from these points using linear interpolation. As another example of interpolated data, we can mention *elevation data* which can be represented by a *Triangulated Irregular Network*, i.e., a finite set of points P along with their elevation. An interpolation based on a triangulation of P gives the value of the interpolated height at any location.

The data model proposes, in the spirit of the constraint data model, to see interpolated pointsets as infinite relations and to express queries on these pointsets

with the standard SQL language. It also includes an accurate finite representation for interpolated data and an algorithm to evaluate queries at a low cost. We currently proceed with the implementation of the model in DEDALE, as explained in the sequel.

We begin with a brief description of the model (a detailed presentation of the constraint model can be found in Chapter 5). All the remainder of the presentation is devoted to the physical aspects of spatio-temporal data management in DEDALE. We first describe the architecture of the prototype and then give detailed explanations on data storage, indexing and query processing. A simple example summarizes the presentation.

7.5.2 Interpolation in the Constraint Model: Representation of Moving Objects

The constraint model can be used efficiently to model both interpolated and non-interpolated data, and allows to query interpolated and non-interpolated data uniformly without the need of specific constructs.

Let us illustrate this with two examples. In case of elevation data, assume that a partition of the plane into triangles T_i is given together with the height of each of the triangles' summits. The interpolated height h of a point p in the plane is defined by first finding i such that T_i contains p. The h value is linearly interpolated from the three heights of the summits of T_i. This latter function depends only upon i and can be defined as a linear function $f_i(x, y)$, valid only for points in T_i. There is a very natural and simple symbolic representation for the three dimensional relation TIN in the linear constraint model:

$$TIN(x, y, h) = \bigvee_i \; t_i(x, y) \wedge h = f_i(x, y)$$

where $t_i(x, y)$ is the semi-linear symbolic representation of the triangle T_i (as a conjunction of three inequalities).

In case of moving objects assume that the position of an object is known at finitely many time points. This defines finitely many time intervals T_i. If the speed of an object is assumed to be constant during each time interval, its position at any time t has coordinates $x = v_i t + x_i$ and $y = w_i t + y_i$ where i is the index of the interval T_i which contains t, v_i, w_i the speed on the axis of the object during that interval, and (x_i, y_i) are chosen appropriately so that the position of the object is correct at the beginning and end of the interval. Thus, the object trajectory $TRAJ(x, y, t)$ can be represented in the linear constraint model as follows:

$$TRAJ(x, y, h) = \bigvee_i \; t_i(t) \wedge x = f_i(t) \wedge y = g_i(t)$$

where $t_i(t)$ are the constraints defining the time interval T_i and f_i, g_i are the linear equations mentioned above. It is easy to see that interpolated relations have similar properties in both cases:

- A subset of attributes which is used as a basis to compute the interpolated values: (x, y) for terrain modeling, t for the trajectories. We denote it as the *key* in the sequel.
- A disjunction of conjuncts, each consisting of equality or inequality constraints restricted to the attributes of the *key*, t_i, and constraints on the interpolated value defined as linear functions on the *key*, f_i and g_i.

This defines a normal form to represent interpolated relations. An important aspect of the model is that interpolated relations can be seen, from the user's point of view, as classical relations, and queried by means of standard query languages. Moreover, it can be shown that evaluating queries upon interpolated databases can be done by manipulating only the key of interpolated relations, while interpolated functions remain identical. A practical consequence is that the cost of query evaluation does not depend on the dimension of the embedding space but on the dimension of the key.

In summary, the model allows to express SQL queries on infinite relations, finitely represented with linear constraints. The formulae which represent objects, such as moving objects, have the specific form outlined above, and this permits to use efficient geometric algorithms for query evaluation.

7.5.3 Architecture

Figure 7.13 depicts the architecture of the DEDALE system. A data server is in charge of data storage and query processing while Java clients propose a graphical interface to express queries and visualize geometric data.

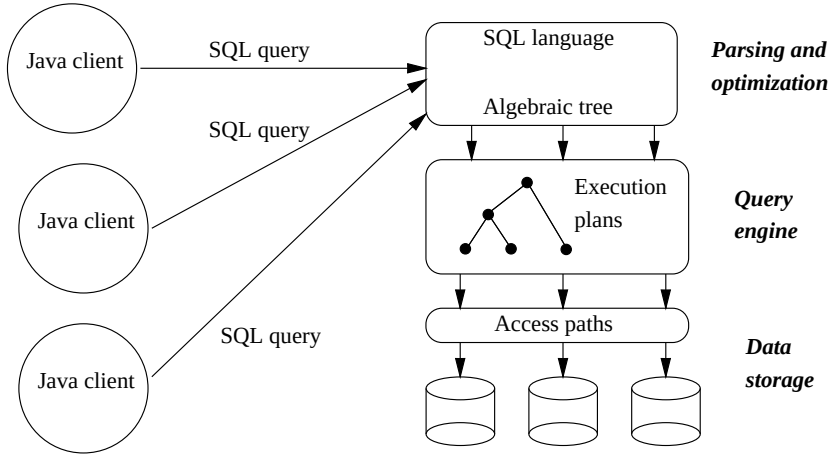

Fig. 7.13. The DEDALEprototype architecture

Our implementation is built on the standard technology for data storage, database indexing, and for the parsing and optimization of the SQL query language.

Since, however, the storage provides a finite representation of conceptually infinitely many points, the query evaluation process features specific algorithms which simulate, upon the finite representation, the semantics of the regular SQL language over infinite sets.

For this part of query processing, we rely on a constraint manipulation engine as already implemented in the first prototype of DEDALE[31]. Note that a vector-based representation could be used as well, but the constraint representation offers a nice framework for the symbolic manipulation of data.

The data storage and query engine levels are built on the BASIS System [34], an extensible C++ implementation of the classical storage management functionalities, as well as an open query processor based on an iteration query execution model, to be described below.

7.5.4 Implementation Details

The implementation mainly aims at showing that the proposed data model for interpolated data can be integrated in a relational DBMS with minimal effort.

Buffer Management and Data Storage. The storage manager provides standard I/O and caching services to access a *database*. A database is a set of binary files which store either datasets (i.e., sequential collection of records) or Spatial Access Methods (SAM). A SAM or index refers to records in an indexed data file through *record identifiers*. Any binary file is divided into *pages* with size chosen at database creation.

The buffer manager handles one or several *buffer pools*. Data files and indexes are assigned to one, global buffer pool. Some operators (e.g. sort and hash) may require a specific buffer pool to the buffer manager. The buffer pool is a constant-size cache with LRU or FIFO replacement policy (LRU by default).

Datasets are stored in sequential files with constant-size records. Since DEDALE proposes a nested data model with only one level of nesting (geometric data are sorted as nested relations), indirection is used to store geometric data. The *main file* stores records with alphanumeric attributes. Spatial attributes are referred to via a `GeomRef` pointer which consists of (i) the bounding box of the spatial object and (ii) the physical address in the *geometric file* where this object can be found.

Although the processing of queries uses a constraint representation for geometric data, at the *data storage* level, we use a vector-based representation. For instance, TINs and trajectories are respectively stored as sequences of triangles and segments in the 3D space. The vector representation is compact and suitable for many basic tasks such as indexing and graphical display. A constant time conversion allows to get a constraint representation from the vector-based representation.

Indexing is based on R*-trees [7]. Since our pointsets are embedded in the 3D space, we could have used a 3D R-tree. However recall that the distinction between the key attributes and the interpolated attributes is semantically meaningful: a query may separately address the multidimensional domains represented

by the key or interpolated variables, respectively. We believe that most queries need only the support of an index on a subset of the variables. We therefore chose to use separate indexing for key attributes and interpolated attributes. For instance, a TIN is indexed on (x, y) by a 2D R*-tree (i.e, we index the bounding rectangle of triangles), and on the h value with a 1D R*-tree (by indexing the intervals). This approach is not the best one in the presence of mobile objects, since rectangles are bad approximations of segments.

The Query Engine. The query engine is based on the pipelined query execution model described in [28]. In this model, query execution is a demand-driven process with iterator functions. Each operator in the query engine is an iterator, and iterators are grouped to form execution plans represented by trees as usual. This allows for a pipelined execution of multiple operations: a tuple goes from one tree node to another, from the disk up toward the root of the query tree, and is never stored in intermediate structures.

Another advantage of the iterator approach is the easy QEP creation by assembling iterators existing in BASIS and the easy extension of BASIS by a trivial integration of a new iterator in its library. All execution plans are *left-deep trees* [28]. In such trees the right operand of a join is always an index, as well as the left operand for the left-most node. The leaves represent data files or indexes, internal nodes represent algebraic operations and edges represent dataflows between operations. Examples of algebraic operations include data access (`FileScan` or `RowAccess`), spatial selections, spatial joins, etc. This simple scheme permits to use the simple indexed nested-loop join algorithm. This strategy is not always optimal, in particular with spatial joins [52,61], but our first goal was model validation rather than query processing optimization.

Query Processing on Interpolated Data. First *disk operators* retrieve the vector representation from files or indexes: "records" are vector-based lists of points. Typical disk operators perform a file or an index scan. They output data into linear constraints, thanks to a trivial conversion. The second category consists of *symbolic operators* which implement the constraint-solving algorithms upon the constraint-based representation. At this level a "record" is a conjunction of linear constraints, representing either a block, or the intermediate structure constructed from several blocks during a join operation.

This design introduces a new level between the physical and abstract representations called *symbolic level* (see Figure 7.14) and based on the linear constraint representation. Note that we could build the system with algorithms working on vector data. However the constraint approach has two advantages. First it provides a uniform representation of interpolated geometric data: both keys and interpolation functions are uniformly represented as linear constraints over a set of variables. Second, such a representation provides a nice support for the specific algorithms, required in our data model, and based on variable substitution.

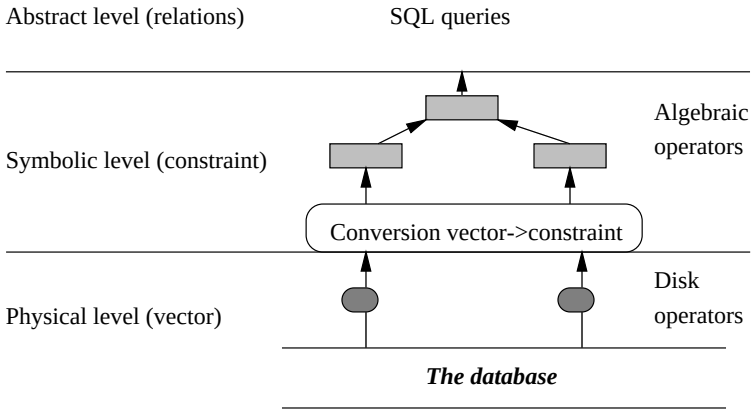

Fig. 7.14. The representation levels

7.5.5 Example of Query Evaluation

In the following, we illustrate the process of query evaluation with the query "*Select in the trajectory of a moving object the points on a TIN with elevation above 10000 meters*". The schema of the database consists of a relation TIN which stores elevation data describing the height above sea, and a relation TRAJ which describes the trajectory of the moving point.

The two point-sets TIN and TRAJ are seen at the query level as classical, although infinite, relations. Hence the query can be expressed with SQL as:

```
SELECT t1.x, t1.y, h
FROM   TIN t1, TRAJ t2
WHERE  t1.x = t2.x AND t1.y = t2.y AND t1.h >= 10000;
```

The data storage consists respectively of a list of points describing (in the 3D space) a trajectory, and a list of triangles describing (also in the 3D space) the TIN. A spatial join between TIN and TRAJ is at the core of the query evaluation. The join is evaluated by an indexed-nested loop join as shown in Figure 7.15. The *trajectory* relation is scanned sequentially, and the bounding box of each segment is used as an argument to carry out a window query on the R-tree indexing the TIN relation. These operations are implemented by *disk* operators. Data is read in a vector format and converted by the join node into a constraint representation. One obtains, at each step, a pair of symbolic tuples of the form shown in left side of Figure 7.16.

The conjunction of constraints from each pair gives a new symbolic tuple with variables x, y, t (see right side of Figure 7.16) as follows. Since x and y are linear functions of t, and h is a linear function of (x, y), a simple substitution of variables is possible. x and y are replaced by the proper function of t in all the inequality constraints. As a result, one obtains x, y, and h as functions of t (which is the *key* of the result), and a somewhat complex conjunction of constraints over the single variable t.

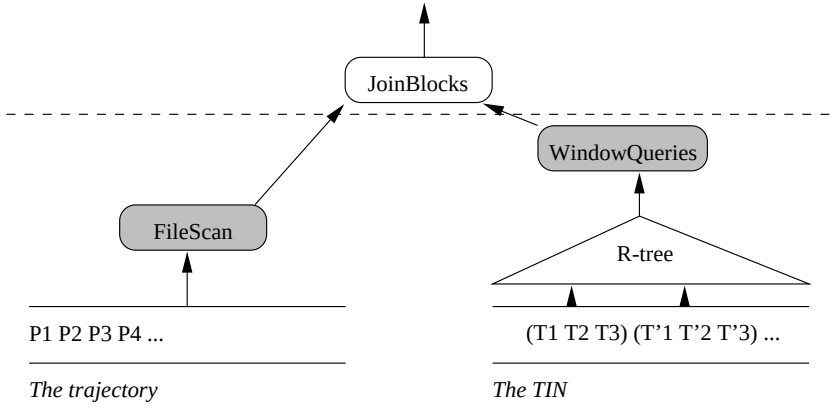

Fig. 7.15. A sample query evaluation

From *TRAJ*	From *TIN*
$0 \leq t \leq 2$	$-4x + y + 1 \leq 0$
$x = 2t + 1$	$y - 3 \leq 0$
$y = t + 2$	$3x + y - 18 \leq 0$
	$h = x + y + 3$

The New Symbolic Tuple
$0 \leq t \leq 2$
$-4(2t + 1) + (t + 2) + 1 \leq 0$
$(t + 2) - 3 \leq 0$
$3(2t + 1) + (t + 2) - 18 \leq 0$
$x = 2t + 1$
$y = t + 2$
$h = (2t + 1) + (t + 2) + 3$

Fig. 7.16. Joining two symbolic tuples

The process ends up by *normalizing* the result, i.e., applying a constraint solving algorithm which checks whether the representation is consistent, and delivers a normalized representation (see [31] for a detailed presentation of the algorithms). This allows to solve the system of equations on t while keeping the other constraints unchanged. The result is depicted in Figure 7.17. For this query, the result is obtained in linear time in the size of this system. The constraint solving algorithm delivers a time interval and linear functions for x, y, and h.

The final result
$0 \leq t \leq 1$
$x = 2t + 1$
$y = t + 2$
$h = 3t + 6$

Fig. 7.17. Final result

In conclusion, the data storage and query engine are currently available, as well as a simplified version of the algorithms presented in [33], restricted to

the manipulation of TINs and mobile objects. The parser and the optimizer are under implementation. Our experimental setting consists of several TINs, ranging from 2000 to 20000 triangles. We plan to use the GSTD generator [76] when it is available to create a consistent sample of mobile objects.

7.6 The Tiger Prototype System

7.6.1 Introduction

TIGER is a temporal database system prototype, which can be extended for spatio-temporal data access [15]. This layered system adopts the dual system architecture (cf. Section 7.2.1). Its salient features include enhanced temporal support through statement modifiers, the intelligent reuse of existing database technology, and the seamless extension of database systems with external modules. Specifically, extensibility is supported by enabling users to plug external modules into the layer. Such modules take spatio-temporal relations as arguments and perform advanced operations on these, the objective being to obtain better overall query performance.

The system is available online via `http://www.cs.auc.dk/~tiger`. Because the web interface uses socket communication, users protected by a firewall might not be able to use it. The source code is therefore also available for downloading and local installation.

7.6.2 Architecture

TIGER follows the dual system architecture. It is implemented as a layer to the Oracle DBMS and systematically enhances that system with temporal functionality. Whenever feasible, the data processing is delegated to the DBMS. However, if functionality and efficiency concerns make the processing by the database system inappropriate, external modules are used instead. External modules seamlessly extend the functionality of the database system. Existing modules perform advanced functions such as coalescing, temporal aggregation, and temporal difference.

The general architecture, depicted in Figure 7.18, identifies three main parts. At the bottom, the DBMS is used as an enhanced storage manager. It is responsi-

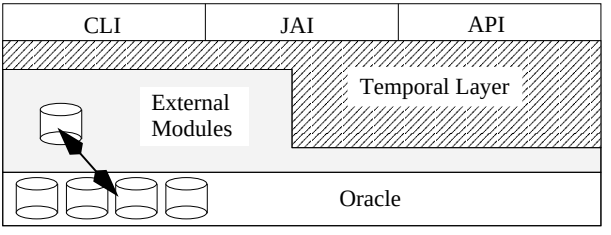

Fig. 7.18. The general layered system architecture of TIGER

ble for all standard database activities. The middle part consists of the temporal layer and the external modules. Together, they provide the core of the temporal functionality. At the top, several interfaces are provided. This includes a command line interface (CLI), a Java applet interface (JAI), and an application program interface (API).

7.6.3 Spatio-temporal Extensions

The TIGER system focuses on temporal database extensions. However it can also be extended for spatio-temporal data access, as can be seen with the query language STSQL [15].

This section briefly sketches the basics of ATSQL—the temporal query language hitherto supported by TIGER. The crucial concept of ATSQL is *statement modifiers*. These control the basic semantics of statements. Four classes of statements are distinguished: upward compatibility, temporal upward compatibility, sequentiality, and non-sequentiality. Each class is described below. For clarity, new syntactic constructs are underlined.

The meaning of a statement modifier naturally divides into four orthogonal parts, namely the specification of the statement class, the time-domain specification, the time-range specification, and the specification of coalescing. We focus on the classes and on coalescing; the reader is referred to [14] for a detailed coverage of domain and range specifications.

Upward Compatibility. It is fundamental that all code without modification will work unchanged with the new spatio-temporal system. A data model is upward compatible with another data model iff all the data structures and legal query expressions of the old model are contained in the new model and iff all queries expressible in the old model evaluate to the same results in the new model. The following statements illustrate upward compatibility:

```
CREATE TABLE p (A INTEGER);
INSERT INTO p VALUES (7);
INSERT INTO p VALUES (8);
COMMIT;
```

These statements are simple legacy SQL–92 statements that must be supported by any reasonable temporal extension of SQL–92. The semantics is the one dictated by SQL–92 [53].

Temporal Upward Compatibility. Temporal upward compatibility, which is easily extended to space [15], aims to ensure a harmonious coexistence of legacy application code and new, temporally-enhanced application code. To illustrate the problem, assume that the new temporal model is in place and that an application needs temporal support, for which reason a snapshot relation must be changed to become a temporal relation. Clearly, it is undesirable (or even impossible) to change the legacy application code that accesses the snapshot relation that has

become temporal. Temporal upward compatibility ensures that this is unnecessary. Essentially, tables can be rendered temporal without changing application code. To illustrate, the statements from the previous section are assumed:

```
ALTER TABLE p ADD VT;
INSERT INTO p VALUES (6);
DELETE FROM p WHERE A = 8;
COMMIT;
SELECT * FROM p;
```

The first statement extends `p` to capture valid time by making it a valid-time table, which contains a timestamping attribute. The insert statement adds 6. Temporal upward compatibility ensures that the past is not changed and that 6 will also be there as time passes by. Similarly, 8 is deleted without changing the past. The select statement returns current, but not past (and future), knowledge. Note that it may not return the valid time.

Sequentiality. Sequentiality protects the investments in programmer training while also providing advanced temporal functionality. Two properties are crucial: snapshot reducibility and interval preservation.

Briefly, snapshot reducibility implies that for all non-temporal queries q, a temporal query q' exists, such that at each snapshot, the result of the temporal query reduces to the result of the original query. For snapshot reducibility to be useful, the relationship between the non-temporal and the temporal query has to be restricted. We require $q' = S_1\, q\, S_2$ where S_1 and S_2 are constant (statement independent) strings. The strings S_1 and S_2 are termed *statement modifiers* because they change the semantics of the entire enclosed statement. ATSQL prepends statements with, e.g., the statement modifier `SEQ VT`:

```
SEQ VT SELECT * FROM p;

CREATE TABLE q (B INTEGER SEQ VT PRIMARY KEY) AS VT;

SET VT PERIOD "1974-1975" INSERT INTO q VALUES (6);
SET VT PERIOD "1976-1978" INSERT INTO q VALUES (6);
SET VT PERIOD "1977-1979" INSERT INTO q VALUES (6);

SEQ VT SELECT * FROM q;
```

The first and last query return all tuples together with their valid time. This corresponds to returning the content of a table at each state. The second statement defines a table `q` and requires column `B` to be a sequenced primary key, i.e., `B` must be a primary key at each state (but not necessarily across states). This constraint implies a conflict between the second and third insert statements: allowing both would violate the primary key constraint for the years 1977 and 1978.

Beyond snapshot reducibility, sequentiality is also preserves the intervals of the argument relations as much as possible in the results. Consider Figure 7.19.

Electricity Bill

Val	VT
150	1993/01–1993/03
150	1993/04–1993/06
70	1993/07–1993/09

Electricity Bill

Val	VT
150	1993/01–1993/06
70	1993/07–1993/09

Fig. 7.19. Snapshot-equivalent relations

Assuming that Val denotes the amount to be paid for electricity during the specified period, the two relations are quite different. Simply merging or splitting the intervals would be incompatible with the intended semantics. Sequentiality preserves intervals as much as possible, i.e., within the bounds of snapshot reducibility [13].

Finally, sequentiality also includes queries of the following type:

```
SEQ VT SELECT * FROM p, q
  WHERE p.X = q.X AND DURATION(VTIME(p),YEAR) > 5;
```

The query is quite natural and easy to understand. It constrains the temporal join to p-tuples with a valid time longer than 5 years. The temporal condition cannot be evaluated on individual snapshots because the timestamp is lost when taking a snapshot of a temporal database, and it thus illustrates how sequentiality extends snapshot reducibility to allow statement modifiers to be applied to all statements [14].

Non-sequentiality. As discussed above, sequenced statements are attractive because they provide built-in temporal semantics based on the view of a database as a sequence of states. However, some queries cannot be expressed as sequenced queries. Therefore, a temporal query language should also allow *non-sequenced* queries with no built-in temporal semantics being enforced. ATSQL uses the modifier `NSEQ VT` to signal non-sequenced semantics, i.e., standard semantics with full explicit control over timestamps:

```
NSEQ VT SELECT * FROM p, q
 WHERE VTIME(p) PRECEDES VTIME(q) AND A = B;
```

The query joins p and q. The join is not performed at each snapshot. Instead, it is required that the valid time of p precedes the valid time of q. A non-temporal relation results.

Coalescing. Coalescing merges tuples into a single tuple if they have overlapping or adjacent timestamps and identical corresponding attribute values [16]. Coalescing is allowed at the levels where the modifiers are also allowed. In addition, as a syntactic shorthand, a coalescing operation is permitted directly after a relation name in the from clause. In this case, the coalesced instance of the relation is considered.

```
SEQ VT SELECT * FROM q;
(SEQ VT SELECT * FROM q)(VT);
SEQ VT SELECT * FROM q(VT);
```

7.6.4 Tiger's Implementation

TIGER is an heterogeneous system that employs different programming paradigms. Specifically, the code is a mixture of C++ (81 KB), C (14 KB), Java (88 KB), and Prolog (160 KB). The implementation of TIGER is based on the architecture of ChronoLog [11], a temporal deductive database system, which uses a layer to translate temporal FOPL to SQL. Another related system is TimeDB [12]. TIGER is the first system that employs external modules, is accessible online and can be used for distance learning.

The temporal layer consists of thirteen modules as illustrated in Figure 7.20. An arrow indicates the direction of an import (e.g., module `meta` imports services from module `unparser`). The layer translates temporal statements to sequences of legacy SQL statements. It adheres to standard compiler implementation techniques [20]. We briefly discuss the prime functionalities associated with the main modules.

Module `interpret` acts as a dispatcher. It calls the scanner and parser to construct a parse tree. A conservative approach is pursued to not break legacy code. Thus, if a statement cannot be identified as being temporal and if the statement does not access temporal data structures, it is passed on to the DBMS. Modules `rewrite` and `deps` normalize and check statements. This includes the rewriting of subqueries, the verification of relation schemas, the lookup of missing table qualifiers, and the detection of dependencies implied by views and integrity constraints. Module `trans` translates temporal queries to sequences of non-temporal ones. For involved temporal statements, this includes the generation of calls to external modules. Data manipulation statements, views, and integrity constraints are handled in the modules `dml`, `views`, and `constraint`, respectively. Module `meta` provides a general-purpose interface to the DBMS and a special-purpose interface to the layer's additional temporal metadata, which is stored in the DBMS.

The purpose of the external modules exists to aid in the computation of queries that the underlying DBMS computes only very inefficiently. For example, although it is possible to perform coalescing, temporal difference, and temporal aggregation in the DBMS, this is exceedingly inefficient [16] and should be left to external modules. External modules fetch the required data from the DBMS,

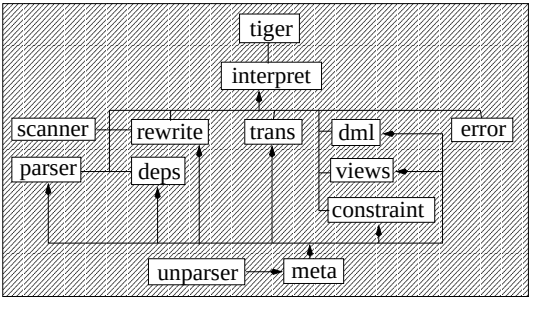

Fig. 7.20. The temporal layer

process it, and store the result back for further processing. To enable the use of external modules, the temporal layer isolates the subtasks that should be delegated to the modules, and an interface between the temporal layer and external modules is provided that consists of a set of procedures of the following form:

```
EM_coal_vt(char* sql, char* table)
EM_coal_tt(char* sql, char* table)
EM_coal_bi_tt(char* sql, char* table)
EM_coal_bi_vt(char* sql, char* table)
EM_diff_vt(char* sql, char* table)
```

All procedures take as input an SQL statement that defines the argument data and return the name of a temporary table that stores the computed result. Additional input parameters can be passed as needed.

The different components of the external module block are shown in Figure 7.21. The `DB` class handles connections to the DBMS. This service is used by

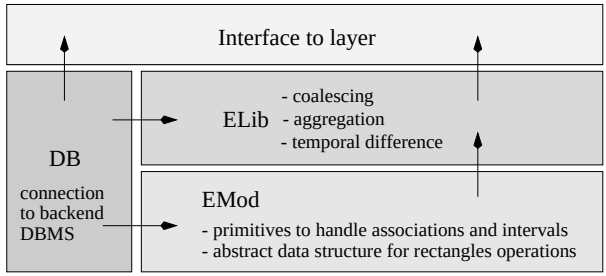

Fig. 7.21. The external modules

all other modules. The `EMod` class implements an abstract matrix structure that is used to implement advanced bitemporal algorithms. Essentially, the matrix keeps track of a set of rectangles. Together with an association structure, the matrix is used to merge rectangles (coalescing) or to split them (difference, aggregation). A coloring scheme is used to ensure interval preservation. The `ELib` class uses the services of the `EMod` and `DB` classes to provide external algorithms for coalescing, difference, and aggregation.

7.6.5 Processing Queries Using External Modules—Case Study

We use the following ATSQL statement to illustrate processing that includes external modules.

```
SEQ VT SELECT r.a
  FROM (SEQ VT SELECT p.a
          FROM p, q
          WHERE p.a = q.a )(VT) AS r
  WHERE VTIME(r) OVERLAPS PERIOD "10-20";
```

An external module is to perform the valid-time coalescing that occurs in the middle of the statement. Thus, the statement has to be split up. We also want to maximally use the underlying DBMS (e.g., for processing joins), thus using only the layer when the DBMS cannot be used with reasonable efficiency.

The innermost part is a sequenced valid-time join, which computes the intersection of overlapping valid times. The innermost part is translated into an SQL statement `Q1` (next), which performs the necessary intersection of the valid times (Oracle's SQL with functions `GREATEST` and `LEAST` is used).

```
Q1 = SELECT GREATEST(P.s,Q.s) s, LEAST(P.e,Q.e) e, P.A A
     FROM P, Q
     WHERE GREATEST(P.s,Q.s)<=LEAST(P.e,Q.e) AND P.A=Q.A;
```

Next, the external module takes over, and `Q1` is passed as the first argument to `EM_coal_vt(char* sql, char* table)` (cf. above), which implements the coalescing operation. An auxiliary table is first created:

```
CREATE TABLE T (s DATETIME, e DATETIME, a INTEGER);
```

Then the tuples identified by `Q1` are retrieved. They are ordered so that coalescing can be performed on the fly [16].

```
SELECT A, s, e
FROM (Q1)
ORDER BY 1, 2, 3;
```

The coalesced tuples are computed and stored in the temporary table, the name of which is finally returned by the external module. The DBMS then takes over and finishes the computation, using the data stored in the temporary table:

```
SELECT R.s s, R.e e, R.A A
FROM (SELECT T.s, T.e, T.A FROM T) R
WHERE r.e > 10 AND 20 > r.s;
```

7.7 The GeoToolKit Prototype System

7.7.1 Introduction

A subsequent analysis of geoscientific and particularly geological domains showed that they use common data sources (3D data, cross sections, digital elevation models etc.) to a large extent, utilizing them for special objectives [3]. They also share a lot of functionality primarily concerning spatial data management and geometric operations. The idea to reuse already implemented components (sources) in the development of new applications was obvious. This was the starting point for designing a "geo" tool kit rather than implementing a series of specialized systems from scratch.

Extensibility in GeoToolKit refers to the extension of the 3D geometric data model classes and the plugging in of new spatial access methods. Contrary to the CONCERT prototype, the temporal access has not yet been integrated into the multi-dimensional access methods. However, GeoToolKit has been extended by temporal classes to support the animation of geological processes.

7.7.2 Architecture

GeoToolKit [3,4] follows the layered system architecture. It has been developed on top of the OODBMS ObjectStore [56]. The system architecture of GeoToolKit is shown in Figure 7.22. It is divided into two main parts: a C++-library and interactive tools. The class library consists of ObjectStore-based classes for spatial data maintenance. The graphical classes provide the visualization of 2D maps and 2D/3D areas which are based on motif. To communicate with external geoscientific tools, first a protocol on top of UNIX-sockets has been realized. The case study on geological basin evolution which is shown below uses this low-level communication. However, an advanced CORBA-based communication architecture has been developed, too.

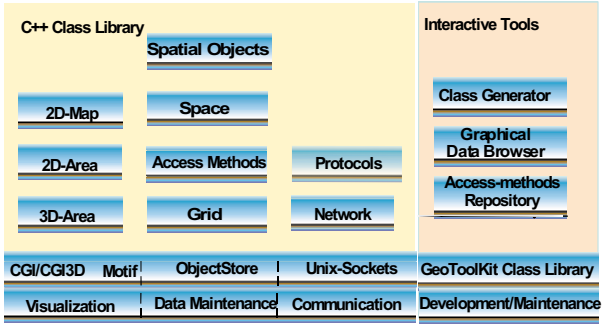

Fig. 7.22. GeoToolKit system architecture

GeoToolKit's architecture has been extended via wrapper technology and a CORBA-based infrastructure to enable remote data and operation access from other 3D geo-information components. The approach has been tested with GOCAD [51,27] a geological 3D modeling and visualization tool. The prototype implementation uses Orbix [49] for transparent network access between the heterogeneous computing platforms. Covered by a wrapper, GeoToolKit can be used as part of a distributed geo-database. The wrapper enables the easy access to the DBMS from other remote CORBA clients. Using CORBA, clients may concurrently access the database on condition that they keep CORBA compatibility. The approach was evaluated and proved by the development of an object database adapter (ODA) prototype and a specialized spatial ODA (SODA). For more detail see [17].

GeoToolKit's interactive tools consists of modules for comfortable loading of databases and navigation through data sources. The class generator provides the generation of user-defined classes without C++-knowledge.

7.7.3 Spatio-temporal Extensions

Spatial Extensions. GeoToolKit[6] primarily deals with two basic notions: a `SpatialObject` and a collection of spatial objects referred to as a `Space`.

On the abstract level a spatial object is defined as a point set in the three-dimensional Euclidean space. Diverse geometric operations can be applied to a spatial object. However, they cannot be implemented unless a spatial object has a concrete representation. There is a direct analogy with the object-oriented modeling capabilities. An abstract spatial object class exclusively specifies the interface inherited by all concrete spatial objects. A concrete object is modeled as a specialization of the abstract spatial object class. It provides an appropriate representation for the object as well as the implementation for the functions.

The geometric functionality involves geometric predicates returning `true` or `false` (e.g. contains), geometric functions (e.g., distance) and geometric operations (e.g. intersection). The geometric operations are algebraically closed. The result of a geometric operation is a spatial object which can be stored in the database or used as an argument in other geometric operations. Naturally every spatial object class includes a set of service facilities required for the correct maintenance of objects (clone, dynamic down cast). The GeoToolKit class hierarchy includes the following classes:

- 0D-3D spatial simplexes: `Point`, `Segment`, `Triangle`, `Tetrahedron`;
- 1D-3D spatial complexes: `Curve`, `Surface`, `Solid`;
- Compound objects: `Group`;
- Analytical objects: `Line`, `Plane`.

Usually complexes are approximated (digitized) and represented as homogeneous collections of simplexes. A `Curve` (1D complex) is approximated through a polyline, a `Surface` and a `Solid` - as a triangle and tetrahedron network, respectively. However, it is not intended to restrict users only with the representations supplied with GeoToolKit. Complex spatial objects are designed in such a way that they do not predefine a physical layout of objects. They contain a reference to a dependent data structure referred to as a representation. The two layer architecture allows for the object to have multiple representations (e.g., one representation for the compact storage, another more redundant one for efficient computations). An object can change its representation without changing the object identity. This feature is of extreme importance in the database context since an object can be referred to from multiple sources. Following certain design patterns a user is able to integrate his own special-purpose representation within GeoToolKit's standard classes.

Spatial objects of different types can be gathered into an heterogeneous collection, called `Group`, which is further treated as a single object. A group is a construction for the representation of the results of geometric operations.

[6] GeoToolKit has been developed in the groups of A. Cremers and A. Siehl in close cooperation with the Geological Institute and SFB 350 at Bonn University, funded by the German Research Foundation.

A space is a special container class capable of efficient retrieval of its elements according to their location in space specified either exactly by a point or by a spatial interval. A spatial interval, often referred to as a bounding box, is defined in 3D-space as a cuboid with the sides parallel to the axes. Since all operations in the Cartesian coordinate system are considerably faster for cuboids than for other objects, the approximation of spatial objects by their bounding boxes is intensively used as effective pre-check by geometric operations and spatial access methods in GeoToolKit.

A space serves both as a container for spatial objects and as a program interface to the spatial query manager which is realized as an internal GeoToolKit library function linked to a geo-application. The spatial query manager is invoked by member functions of the class `Space`. Practically, a call of any space method means a call of the spatial query manager. A user can add/remove a spatial object to/from a space syntactically in the same way as in the case of usual object collections. However, all changes will go through the spatial query manager. A spatial retrieval is performed through the family of *retrieve* methods. The task of various retrieve methods is to provide a convenient interface for the spatial query manager. In the simplest case a retrieve member function takes a bounding box as a parameter and returns a set of spatial objects contained in or intersected by this bounding box. A user can also formulate his query in the ObjectStore style as a character string. Such interface may be useful for the implementation of interactive retrieval. However, it needs multiple conversions of data and therefore it is not convenient for the internal use within a program. A spatial retrieval involving "indirect" spatial predicates (e.g. intersects) is usually decomposed into two sequential steps. In the first step the query manager retrieves all objects which intersect the bounding box of the given object. On the second step it checks whether the pre-selected objects really intersect the given object.

To provide efficient retrieval, a space must have a spatial index. In order to make an arbitrary user-defined index known to GeoToolKit, it must fit the interface defined within the abstract `AccessMethod` class. GeoToolKit supports two indexing methods: the R-Tree [39] for a pure spatial retrieval and the LSD-Tree [42] for a mixed spatial/thematic retrieval.

Temporal Extensions. The ability to follow a topological evolution of geological entities is of special interest for geo-scientists. Geological entities are characterized by relative large and irregular time intervals between time states available. On the contrary, for the smooth animation small regular time intervals are required. Therefore missing time states need to be interpolated. An interpolation between primitive simplex objects (simplexes) is straightforward. An interpolation of complexes can be reduced to the simplex-to-simplex interpolation only if complexes have the same cardinality. However, an object may change its size and/or shape with respect to time in such a way that it will need more simplexes for the adequate representation. For example, in the result of deformations a flat platform (for the representation of which two triangles are quite enough) may transform into a spherical surface which will need a much larger number of

triangles for the qualitative representation. GeoDeform [1] is a geo-scientific application for calculating geological deformations which have been developed at the Geological Institute of the Bonn University. For the visualization of spatial objects changing in time GeoDeform uses a model proposed in the graphical library GRAPE [62]. According to this model, each time state contains two representations of the same object with different number of simplexes (a discretization factor). The first representation (post-discretization) corresponds to the approximation of the current state of the object with the discretization required by its current size and shape. The second one (pre-discretization) corresponds to the approximation of the current state of the object but with the discretization used in the previous state. Due to this extension an interpolation can always be performed between representations with the same discretization factor: the post-discretization of the previous state and the pre-discretization of the current state of the object.

To provide an appropriate maintenance of a large number of spatio-temporal objects, GeoDeform was extended with a database component developed on the GeoToolKit basis. The task was to integrate into GeoToolKit's pure spatial classes a concept of time so that a spatial functionality already available could be re-utilized and a maximal level of compatibility with GRAPE was provided. To represent different time states of the same spatial object, we introduced a class `TimeStep` (see Figure 7.23), which contains a time tag and two references (pre and post) to spatial objects. If the pre- and post-discretization factors are equal, pre and post links simply refer to the same spatial object. The model proposed is general enough since an arbitrary spatial object can be chosen as a representation of a time state. In the case of GeoDeform there are geological strata and faults modeled through GeoStore's Stratum and Fault classes which in turn are defined as a specialization of GeoToolKit's `Surface` class. A sequence of `TimeStep` instances characterizing different states of the same spatial object are gathered into a spatio-temporal object (class `TimeSequence`). Being a specialization of the abstract class `SpatialObject`, an instance of `TimeScene` can be treated in the same way as any other spatial object, i.e. it can be inserted into a space as well as participate in all geometric operations. The spatial

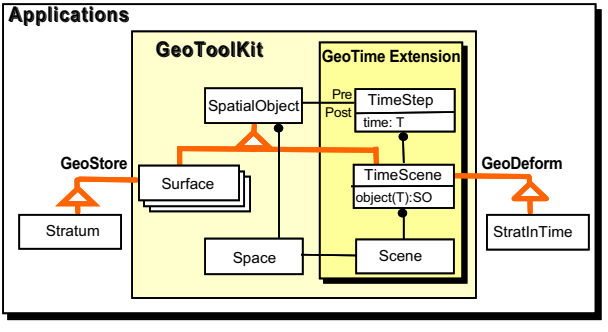

Fig. 7.23. Temporal extensions for GeoToolKit

functionality is delegated by default to the spatial object referred to in the latest `TimeStep` instance. A selection for the interpolation differs from a common selection with a specified key. If there is no object in the *time sequence* that hits exactly the time stamp t specified in the retrieve function, instead of NULL it should return a pair of neighbor time steps with time tags t1 and t2, so that $t1 < t < t2$. The same is valid for the time interval. If the interval's margins do not exactly hit the time step instances in the sequence, the resulting set includes all time steps fitting the interval completely extended with the nearest ancestor of the time step with the lowest time tag and the nearest successor of the time step with the highest time tag. Any `TimeSequence` instance can be inserted in and spatially retrieved from the GeoToolKit's space as any other spatial object. However, to perform a temporal retrieval, a special container class `Scene` is introduced. This class is capable of both spatial and temporal retrieval.

7.7.4 Implementation Details

GeoToolKit is open for new spatial and temporal indexing methods. A general technique is developed which allows the integration of arbitrary spatial indexing methods within the object-oriented DBMS ObjectStore. A new container type for the maintenance of spatial objects (space) has to be defined and for this class a spatial query manager has to be implemented. The spatial query manager overloads the OODBMS native query manager. It parses a query, extracts spatial predicates and checks whether a spatial index is available. If no spatial index is associated with the space, it simply forwards the query to the native query manager. If a spatial index is found, the spatial query performs index-based retrieval. The results (if not empty) together with the rest of the query are forwarded to the native query manager. Since the spatial query manager performs the role of a preprocessor, the syntax of the native query language can be preserved or even extended.

To enable the cooperation between the spatial query manager and spatial indexing methods, they must have a common interface. This requirement is not as restrictive as it seems to be at the first glance because the majority of the spatial indexes exclusively deals with the bounding box approximation of spatial objects. Index developers do not even need to modify the sources to fit the required interface. A usual technique is to develop an adapter class which carries out all necessary conversions and then it simply calls corresponding functions according to the pre-defined interface.

The situation is more complicated in the case of so-called combined indexing. Combined indexes may be beneficial when separately neither a spatial nor a thematic component of the search criterion is selective enough. Distinct to the bounding box, the number and types of non-spatial attributes cannot be predefined. To combine spatial and heterogeneous non-spatial subkeys, GeoToolKit offers a special construction. The multikey class provides a uniform access to an arbitrary subkey according to its number (dimension). The retrieval with the multikey is similar to the spatial retrieval with the only difference that a generic multikey substitutes the bounding box.

A query can be formulated and called either in the same way as a usual ObjectStore query or by means of the additional retrieve function. The retrieve function demands from the user to separate explicitly the spatial and non-spatial parts of the query. The spatial part is represented as a bounding box or as a multidimensional key. If a space has more than one spatial index and a user wants a particular index to be used for the retrieval, he can explicitly specify it in the retrieve function.

To maintain the indexes consistently within GeoToolKit, every member function (dealing with spatial or temporal updates) incorporates additional checks before and after performing updates. The pre-check function tests whether an object is contained in at least one indexed space. If yes, the bounding box is stored. The post-check function activates re-indexing only when the bounding box has been changed. To eliminate re-indexing by serial updates, GeoToolKit provides an update block.

7.7.5 Case Studies

GeoToolKit has been tested with different geological applications like well management, 3D geological modeling and geological restoration based on time-dependent geometries [1]. Figure 7.24 shows three time steps of a basin evolution modeled with GeoDeform coupled with GeoToolKit. The figure shows a perspective view of the southern part of the Lower Rhine Basin, Germany, towards the Northeast with the base of the Oligocene and synthetic as well as antithetic faults. Black arrows indicate direction and cumulative amount of extension, whereas negative numbers in *italics* show subsidence at selected points.

In GeoDeform's concept of geological time-space modeling [62], a 4D model consists of a set of states of modeled 3D objects at different times. The time steps between the object states can be different for each object chain and need not to be equidistant. Interpolation (e.g., linear) can be done between known object states. A "time frame" representation of the 4D model is the set of object states at a distinct system time. Time-dependent 3D objects can be modified, added to, or deleted from the 4D model. The possibility to handle varying discretization and topology for each object at each time step makes the concept very flexible for interactive geological modeling and visualization. From the 4D model, a time schedule of subsidence and deformation can be deduced and calibrated against the geological observations, providing new insights into the origin of the basin (Figure 7.24).

Examples for temporal and spatio-temporal queries are:

- `os_Set<TimeStep>tmp=scene.retrieve(time>-20.0 && time<0);`
- `os_Set<TimeStep>tmp=scene.retrieve(bb,time>-20.0 && time<0);`

The first query yields all states of geological objects in a scene which existed 20 million years ago until the recent state. The second one additionally selects the geological objects which are located within a specified bounding box.

The development of GeoDeform, a database component coupled with GeoToolKit, proved the advantages of the tool kit approach. GeoDeform classes

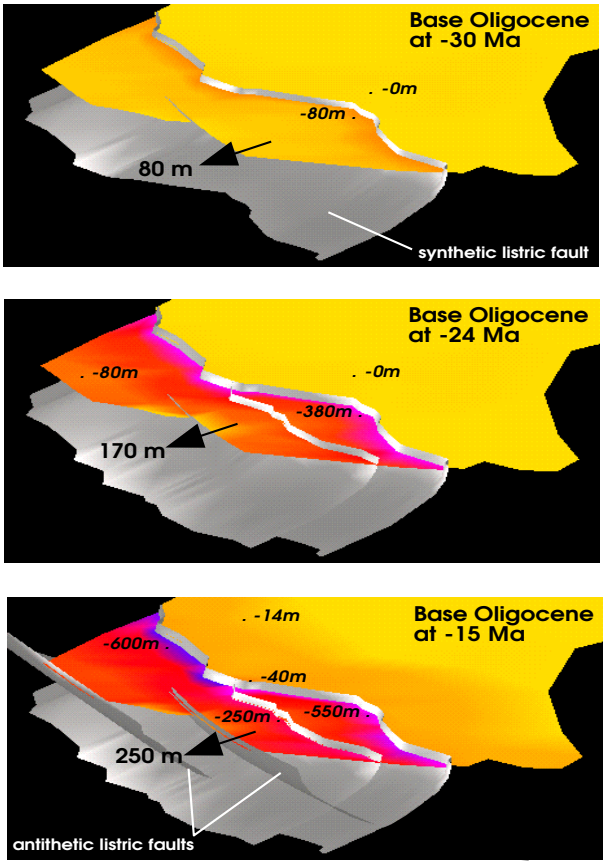

Fig. 7.24. Three time steps of basin evolution in the Lower Rhine Basin, Germany, modeled by GeoDeform coupled with GeoToolKit [1]

contain now only geology-specific members. Geometric relationships between geometric objects are hidden within GeoToolKit. However, the GeoToolKit functionality (spatial retrieval, indexing, etc.) is still fully available for the extended objects.

The developers can focus on application semantics instead of optimally assembling spatial objects from multiple relational tables or the re-implementation of routine geometric algorithms. Classes designed for particular applications can either be used directly or can be refined for other applications.

7.8 Conclusions

We conclude this chapter with a brief summary of the architectures of the prototype systems and their contributions to spatio-temporal data management.

System	Architecture	Main Contribution
CONCERT	extensible	generic index support
SECONDO	extensible	framework for spatio-temporal data types
DEDALE	extensible	constraint model with handling of interpolated spatial data
TIGER	layered RDBMS	(spatio-)/temporal query language
GEOTOOLKIT	layered OODBMS	geometric 3D/4D data types and indexes

Fig. 7.25. Comparison of the prototype systems

System Architectures. As depicted in Figure 7.25, CONCERT, SECONDO, and DEDALE are representatives of the extensible system architecture. These systems allow a more or less seamless extension towards spatio-temporal database systems. CONCERT's extensibility aims at a flexible index support for arbitrary data stores, whereas SECONDO supports the design of spatio-temporal data types on top of a data store. DEDALE's data storage and query engine consist of an extensible C++-implementation. TIGER is a representative of the layered system architecture. Its temporal database query language is based on relational database technology. GEOTOOLKIT is closely coupled with the OODBMS ObjectStore, which basically serves as an ordinary data store. GEOTOOLKIT allows to extend ObjectStore with spatio-temporal data types and access methods. In this sense, GEOTOOLKIT is a representative of the extensible system architecture. However, one can also argue that GEOTOOLKIT is build on top of ObjectStore and thus is a representative of the layered system architecture.

Specific Contributions to Spatio-temporal Data Management. CONCERT provides a framework of a generic index, which can be directly used for the refinement of specific spatio-temporal data types and access methods. This index has several properties:

- *It is generic in the sense of data types.* It is valid and usable for all data types following the `SPATIAL` concept of CONCERT independent of their location or size.
- *It is generic in the sense of tree behavior.* Changing the tree heuristics leads to different tree derivations.
- *It is derived from the main issue of each data management system — the data.* In contrast, other approaches usually derive their genericity by generalizing algorithms or methods.

Throughout the whole code of CONCERT's 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., as 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 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.

SECONDO has been introduced as a generic development environment for non-standard multi-user database systems. At the bottom architecture level, SECONDO offers tools for efficient and comfortable handling of nested lists and catalogs, a simplified interface to the underlying SHORE storage manager, a tool for efficient management of tuples with embedded large objects, and an SOS compiler. Algebra modules for standard and relational data types and operators as well as simple user interface clients have been implemented. The core part of SECONDO, its extensible query processor, is characterized by the following highlights:

- Its formal basis to describe a generic query plan algebra giving a clear algorithm for translating a query plan into an operator tree.
- Functional abstraction is a well-defined concept in SOS. This leads to a very clean, simple, and general treatment of parameter expressions of operators.
- `stream` is a built-in type constructor in SECONDO. Simply writing the keyword `stream` in the type mapping of an operator lets the query processor automatically set up calls of the evaluation function for this operator in stream mode. For this reason, SECONDO can uniformly handle streams of anything, not just tuples. Also, a query plan can freely mix stream and non-stream operators.
- SECONDO includes complete type checking, type mapping, and resolution of operator overloading.

DEDALE contributes to spatio-temporal databases by its extension to handle interpolated spatial data. Its model covers moving objects like trajectories and interpolated spatial data like Triangulated Irregular Networks. The constraint data model sees interpolated pointsets as infinite relations and expresses queries on these pointsets with the standard SQL language. It also provides an accurate finite representation for interpolated data. Furthermore, an algorithm to evaluate queries at a low cost is supplied.

TIGER is a temporal database system that demonstrates the use of ATSQL's statement modifiers to manage temporal information. It can also be extended to spatio-temporal database access. It features an enhanced support for the time dimension through ATSQL. The highlights concerning the TIGER implementation are:

- the intelligent reuse of existing database technology,
- a seamless extension of database systems with external modules, and
- an applet interface that supports distance learning as application of TIGER.

Emphasizing the temporal management of data, TIGER counterbalances the other prototype systems discussed in this chapter, which primarily focus on the spatial aspects of data management.

GEOTOOLKIT is a spatio-temporal extension of the OODBMS ObjectStore. It has been designed to support especially geological database applications which are intrinsically space (3D) and time-dependent (4D). The highlights of the GEOTOOLKIT implementation are:

- 3D geometric data types based on simplicial complexes,
- advanced 3D geometric algorithms,
- extensions for temporal data handling,
- coupling with a 3D visualization tool, and
- an open CORBA-based system architecture.

GEOTOOLKIT's geometric 3D data types, especially the triangle and tetrahedron networks, allow advanced spatial database queries like the intersection of a 3D-object with a set of other 3D-objects. The temporal extensions of GEOTOOLKIT provides the selection of snapshots of 3D objects and an adequate visualization.

GEOTOOLKIT has been tested during the last years by different geological groups at Bonn University. The most advanced application on top of GEOTOOLKIT is GEOSTORE, an information system for geologically defined 3D geometries in the Lower Rhine Basin, Germany. Furthermore, a data management tool and a spatio-temporal data browser have been developed.

In summary, although none of the prototype systems presented in this chapter are complete spatio-temporal database management system, they implement important issues of spatial and temporal database management systems and thereby provide the platform for future systems with full spatio-temporal support. There is also the legitimate hope that these prototype systems could be predecessors of components and services to be integrated in an overall spatio-temporal system architecture.

References

1. R. Alms, O. Balovnev, M. Breunig, A.B. Cremers, T. Jentzsch, and A. Siehl. Space-Time Modelling of the Lower Rhine Basin Supported by an Object-Oriented Database. *Physics and Chemistry of the Earth*, 23(3):251–260, 1998.
2. ARC/INFO. ArcInfo 8, a New Standard in Professional GIS, ESRI. Brochure `http://www.esri.com/library/brochures/pdfs/arcinfo8ad.pdf`, ESRI, 2000.
3. O. Balovnev, M. Breunig, and A.B. Cremers. From GeoStore to GeoToolKit: The Second Step. In M. Scholl and A. Voisard, eds., *Advances in Spatial Databases, Proc. 5th Int. Symposium, SSD'97*, LNCS, Vol. 1262, pp. 223–237, Springer-Verlag, 1997.
4. O. Balovnev, M. Breunig, A.B. Cremers, and M. Pant. Building Geo-Scientific Applications on Top of GeoToolKit: a case study of Data Integration. In *Proc. 10th Int. Conf. on Scientific and Statistical Database Management*, pp. 260–269, IEEE Computer Science Press, 1998.

5. D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988.
6. L. Becker and R.H. Güting. The GraphDB Algebra: Specification of Advanced Data Models with Second-Order Signature. Informatik-Report 183, FernUniversität Hagen, Germany, 1995.
7. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In H. Garcia-Molina and H. Jagadish, eds., *Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 19, No. 2, pp. 322–331, ACM Press, 1990.
8. S. Blott, H. Kaufmann, L. Relly, and H.-J. Schek. Buffering Long Externally-Defined Objects. In *Persistent Object Systems, Proc. 6th Int. Workshop*, Workshops in Computing, pp. 40–53, Springer-Verlag, 1995.
9. S. Blott, L. Relly, and H.-J. Schek. An Abstract-Object Storage Manager. In H. V. Jagadish and I. S. Mumick, eds., *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 25, No. 2, pp. 330–340, ACM Press, 1996.
10. T. Bode, A.B. Cremers, and J. Freitag. OMS – An Extensible Object Management System. In R. Bayer, Härder, and P. C. Lockemann, eds., *Objektbanken für Experten*, pp. 29–54, Informatik aktuell, Springer-Verlag, 1992.
11. M.H. Böhlen. *Managing Temporal Knowledge in Deductive Databases*. PhD thesis, Eidgenössisch Technische Hochschule (ETH) Zürich, Switzerland, 1994.
12. M.H. Böhlen. Temporal Database System Implementations. *ACM SIGMOD Record*, 24(4):53–60, 1995.
13. M.H. Böhlen, R. Busatto, and C.S. Jensen. Point-Versus Interval-Based Temporal Data Models. In *Proc. 14th IEEE Int. Conf. on Data Engineering, ICDE'98*, pp. 192–200, IEEE Computer Society Press, 1998.
14. M.H. Böhlen and C.S. Jensen. Seamless Integration of Time into SQL. Technical Report R-96-2049, Department of Computer Science, Aalborg University, Denmark, 1996.
15. M.H. Böhlen, C.S. Jensen, and B. Skjellaug. Spatio-Temporal Database Support for Legacy Applications. In L. Haas and A. Tiwary, eds., *SIGMOD'98, Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 25, No. 2, pp. 226–234, ACM Press, 1998.
16. M.H. Böhlen, R.T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, eds., *Proc. 22nd Int. Conf. on Very Large Data Bases, VLDB'96*, pp. 180–191, Morgan Kaufmann, 1996.
17. M. Breunig, A.B. Cremers, H.-J. Götze, S. Schmidt, R. Seidemann, S. Shumilov, and A. Siehl. First Steps Towards an Interoperable GIS - An Example from Southern Lower Saxony. *Physics and Chemistry of the Earth*, 24(3):179–189, 1999.
18. M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E. J. Shekita. The Architecture of the EXODUS Extensible DBMS. In K.R. Dittrich and U. Dayal, eds., *Proc. 1st Int. Workshop on Object-Oriented Database Systems*, pp. 52–65, IEEE Computer Society Press, 1986.
19. M.J. Carey, D.J. DeWitt, M.J. Franklin, N.E. Hall, M.L. McAuliffe, J.F. Naughton, D.T. Schuh, M.H. Solomon, C.K. Tan, O.G. Tsatalos, S.J. White, and M.J. Zwilling. Shoring Up Persistent Applications. In *Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 23, No. 2, pp. 383–394, ACM Press, 1994.

20. W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 3 edition, 1987.
21. L. Comet. The Ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
22. S. Dieker and R.H. Güting. Plug and Play with Query Algebras: SECONDO – A Generic DBMS Development Environment. Informatik-Report 249, FernUniversität Hagen, Germany, 1999. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS 2000), September 2000*.
23. S. Dieker and R.H. Güting. Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering*, 32(3):247–269, 2000.
24. S. Dieker, R.H. Güting, and M. Rodríguez Luaces. A Tool for Nesting and Clustering Large Objects. Informatik-Report 265, FernUniversität Hagen, Germany, 2000. In *Proceedings of the 12th International Conference on Scientific and Statistical Database Management, July 2000*.
25. EcoWin, Hanson & Partners, Gothenburg, Sweden. *EcoWin Time Series Extender*, 1999. `http://www.ecowin.com`.
26. R.A. Finkel and J.L. Bently. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.
27. GOCAD Techn. Documentation, 2000. `http://www.ensg.u-nancy.fr/GOCAD`.
28. G. Graefe. Query Evaluation Techniques For Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
29. G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.
30. G. Graefe and W.J. McLenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In A. Elmagarmid and E. Neuhold, eds., *Proc. 9th IEEE Int. Conf. on Data Engineering, ICDE'93*, pp. 209–218, IEEE Computer Society Press, 1993.
31. S. Grumbach, P. Rigaux, M. Scholl, and L. Segoufin. The DEDALE/ Prototype. In G. Kuper, L. Libkin, and J. Paradaens, eds., *Constraint Database Systems*, pp. 365–382, Springer-Verlag, 2000.
32. S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE/ System for Complex Spatial Queries. In L. Haas and A. Tiwary, eds., *SIGMOD'98, Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 25, No. 2, pp. 213–224, ACM Press, 1998.
33. S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating Interpolated Data is Easier than you Thought. *Proceedings of VLDB 2000*, pp. 156-165, Cairo, Egypt, September 2000.
34. C. Gurret, Y. Manolopoulos, A. Papadopoulos, and P. Rigaux. The BASIS System: A Benchmarking Approach for Spatial Index Structures. In M.H. Böhlen, C.S. Jensen, and M. Scholl, eds., *Proc. of the Int. Workshop on Spatio-Temporal Database Management*, LNCS, Vol. 1678, pp. 152–170, Springer-Verlag, 1999.
35. R.H. Güting. Gral: An Extensible Relational Database System for Geometric Applications. In P.M.G. Apers and G. Wiederhold, eds., *Proc. 15th Int. Conf. on Very Large Data Bases, VLDB'89*, pp. 33–44, Morgan Kaufmann, 1989.
36. R.H. Güting. Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. In P. Buneman and S. Jajodia, eds., *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 22, No. 2, pp. 277–286, ACM Press, 1993.
37. R.H. Güting. GraphDB: Modeling and Querying Graphs in Databases. In J. B. Bocca, Matthias Jarke, and C. Zaniolo, eds., *Proc. 20th Int. Conf. on Very Large Data Bases, VLDB'94*, pp. 297–308, Morgan Kaufmann, 1994.

38. R.H. Güting, S. Dieker, C. Freundorfer, L. Becker, and H. Schenk. SECONDO/QP: Implementation of a Generic Query Processor. In T.J.M. Bench-Capon, G. Soda, and A.M. Tjoa, eds., *Database and Expert Systems Applications, Proc. 10th Int. Conf., DEXA'99*, LNCS, Vol. 1677, pp. 66–87, Springer-Verlag, 1999.
39. A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In B. Yormark, ed., *Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 14, No. 2, pp. 47–57, ACM Press, 1984.
40. 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(1):143–160, 1990.
41. J. M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In U. Dayal, P.M.D. Gray, and S. Nishio, eds., *Proc. 21st Int. Conf. on Very Large Data Bases, VLDB'95*, pp. 562–573, Morgan Kaufmann, 1995.
42. A. Henrich, H.-W. Six, and P. Widmayer. The LSD Tree: Spatial Access to Multidimensional Point and Non-Point Objects. In P.M.G. Apers and G. Wiederhold, eds., *Proc. 15th Int. Conf. on Very Large Data Bases, VLDB'89*, pp. 45–54, Morgan Kaufmann, 1989.
43. Informix Software, Inc., Menlo Park, CA. *Excalibur Text Search DataBlade Module: User's Guide, Version 1.1*, 1997.
44. Informix Software, Inc., Menlo Park, CA. *INFORMIX Geodetic DataBlade Module: User's Guide, Version 2.1*, 1997.
45. Informix Software, Inc., Menlo Park, CA. *INFORMIX Spatial DataBlade Module: User's Guide, Version 2.2*, 1997.
46. Informix Software, Inc., Menlo Park, CA. *INFORMIX TimeSeries DataBlade Module: User's Guide, Version 3.1*, 1997.
47. Informix Software, Inc., Menlo Park, CA. *Extending INFORMIX-Universal Server: Data Types, Version 9.1*, 1998.
48. International Organization for Standardization & American National Standards Institute, ANSI/ISO/IEC 9075-2:99. *ISO International Standard: Database Language SQL - Part 2: Foundation)*, September 1999.
49. IONA Technologies Ltd. Orbix Programmers's Guide, Version 2.3, 1997.
50. H. Luttermann and A. Blobel. Chronos: A Spatiotemporal Data Server for a GIS. In *Proc. 9th. Int. Symposium on Computer Science in Environmental Protection*, pp. 135–142, Metropolis, 1995.
51. J.L. Mallet. GOCAD: A Computer Aided Design Program for Geological Applications. In A. K. Turner, ed., *Three-Dimensional Modeling with Geoscientific Information Systems*, pp. 123–142, Kluwer Academic Publishers, 1992.
52. N. Mamoulis and D. Papadias. Integration of Spatial Join Algorithms for Joining Multiple Inputs. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, eds., *Proc. 1999 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, Vol. 28, No. 2, pp. 1–12, ACM Press, 1999.
53. J. Melton and A.R. Simon. *Understanding the New SQL — A Complete Guide.* Morgan Kaufmann, 1993.
54. S. Morehouse. A Geo-Relational Model for Spatial Information. In *Proceedings of Auto Carto 7*, pp. 338–357, 1985.
55. J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.
56. ObjectStore – Online Product Documentation. `http://www.odi.com`.

57. J. Ong, D. Fogg, and M. Stonebraker. Implementation of Data Abstraction in the Relational Database System Ingres. *SIGMOD Record*, 14(1):1–14, 1984.
58. Oracle Corporation. *Oracle8i Spatial: User's Guide and Reference, Release 8.1.6*, 1999.
59. Oracle Corporation. *Oracle8i TimeSeries: User's Guide, Release 8.1.6*, 1999.
60. Oracle Corporation – Product Documentation. `http://www.oracle.com`.
61. A. Papadopoulos, P. Rigaux, and M. Scholl. A Performance Evaluation of Spatial Join Processing Strategies. In R.H. Güting, D. Papadias, and F.H. Lochovsky, eds., *Advances in Spatial Databases, Proc. 6th Int. Symposium, SSD'99*, LNCS, Vol. 1651, pp. 286–307, Springer-Verlag, 1999.
62. K. Polthier and M. Rumpf. A Concept for Time-Dependent Processes. In M. Göbel, H. Müller, and B. Urban, eds., *Visualization in Scientific Computing*, pp. 137–153, Springer-Verlag, 1995.
63. L. Relly. *Open Storage Systems: Pysical Database Design for External Objects*. PhD thesis, Eidgenössisch Technische Hochschule (ETH) Zürich, ETH-Zentrum, CH-8092 Zürich, Switzerland, 1999. (In German).
64. L. Relly and U. Röhm. Plug and Play: Interoperability in CONCERT. In A. Vckovski, K.E. Brassel, and H.-J. Schek, eds., *Interoperating Geographic Information Systems, Proc. 2nd Int. Conf., INTEROP'99*, LNCS, Vol. 1580, pp. 277–291, Springer-Verlag, 1999.
65. L. Relly, H.-J. Schek, O. Henricsson, and S. Nebiker. Physical Database Design for Raster Images in Concert. In M. Scholl and A. Voisard, eds., *Advances in Spatial Databases, Proc. 5th Int. Symposium, SSD'97*, LNCS, Vol. 1262, pp. 259–279, Springer-Verlag, 1997.
66. L. Relly, H. Schuldt, and H.-J. Schek. Exporting Database Functionality — The Concert Way. *Bulletin of the IEEE Technical Committee on Data Engineering*, 21(3):43–51, 1998.
67. B. Salzberg and V.J. Tsotras. A Comparison of Access Methods for Temporal Data. TimeCenter Technical Report TR-18, TimeCenter, 1997.
68. H.-J. Schek, H.-B. Paul, and M.H. Scholl. The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):25–43, 1990.
69. H.-J. Schek and W. Waterfeld. A Database Kernel System for Geoscientific Applications. In D. Marble, ed., *Proc. of the 2nd Symposium on Spatial Data Handling*, pp. 273–288, 1986.
70. P. Seshadri, M. Livny, and R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, eds., *Proc. 22nd Int. Conf. on Very Large Data Bases, VLDB'96*, pp. 99–110, Morgan Kaufmann, 1996.
71. P. Seshadri, M. Livny, and R. Ramakrishnan. The Case for Enhanced Abstract Datatypes. In M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., *Proc. 23rd Int. Conf. on Very Large Data Bases, VLDB'97*, pp. 66–75, Morgan Kaufmann, 1997.
72. Smallworld. SMALLWORLD, the Geographical Information System SMALLWORLD GIS. SMALLWORLD Report, SMALLWORLD Systems GmbH, Ratingen, Germany, 2000.
73. M. Stonebraker. Inclusion of New Types in Relational Database Systems. In G. Wiederhold, ed., *Proc. 2nd IEEE Int. Conf. on Data Engineering, ICDE'86*, pp. 262–269, IEEE Computer Society Press, 1986.

74. M. Stonebraker and L. A. Rowe. The Design of POSTGRES. In C. Zaniolo, ed., *Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, D.C.*, ACM SIGMOD Record, Vol. 15, No. 2, pp. 340–355, ACM Press, 1986.
75. M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.
76. Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento. On the Generation of Spatiotemporal Datasets. In R.H. Güting, D. Papadias, and F.H. Lochovsky, eds., *Advances in Spatial Databases, Proc. 6th Int. Symposium, SSD'99*, LNCS, Vol. 1651, pp. 147–164, Springer-Verlag, 1999.
77. P.F. Wilms, P.M. Schwarz, H.-J. Schek, and L.M. Haas. Incorporating Data Types in an Extensible Database Architecture. In C. Beeri, J W. Schmidt, and U. Dayal, eds., *Proc. 3rd Int. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pp. 180–192, Morgan Kaufmann, 1988.