Object-Relational Spatial Indexing

Hans-Peter Kriegel
University of Munich
Institute for Computer Science
Unit for Database Systems
Oettingenstraße 67, 80538 Munich, Germany
Phone: +49-89-2180-9191
Fax: +49-89-2180-9192
Email: kriegel@dbs.informatik.uni-muenchen.de

Martin Pfeifle
University of Munich
Institute for Computer Science
Unit for Database Systems
Oettingenstraße 67, 80538 Munich, Germany
Phone: +49-89-2180-9331
Fax: +49-89-2180-9192
Email: pfeifle@dbs.informatik.uni-muenchen.de

Marco Pötke
software design & management AG
Carl-Wery-Str. 42, 81739 Munich, Germany
Phone: +49-89-63812-453
Fax: +49-89-63812-490
Email: marco.poetke@sdm.de

Thomas Seidl
RWTH Aachen University
Department of Computer Science IX
Ahornstraße 55, 52056 Aachen, Germany
Phone: +49-241-80-21901
Fax: +49-241-80-22931
Email: seidl@informatik.rwth-aachen.de

Jost Enderle
RWTH Aachen University
Department of Computer Science IX
Ahornstraße 55, 52056 Aachen, Germany
Phone: +49-241-80-21907
Fax: +49-241-80-22931
Email: enderle@informatik.rwth-aachen.de
Object-Relational Spatial Indexing

Abstract: In order to generate efficient execution plans for queries comprising spatial data types and predicates, the database system has to be equipped with appropriate index structures, query processing methods, and optimization rules. Although available extensible indexing frameworks provide a gateway for seamless integration of spatial access methods into the standard process of query optimization and execution, they do not facilitate the actual implementation of the spatial access method itself. An internal enhancement of the database kernel is usually not an option for database developers. The embedding of a custom block-oriented index structure into concurrency control, recovery services and buffer management would cause extensive implementation efforts and maintenance cost, at the risk of weakening the reliability of the entire system. The server stability can be preserved by delegating index operations to an external process, but this approach induces severe performance bottlenecks due to context switches and inter-process communication. Therefore, we present the paradigm of object-relational spatial access methods that perfectly fits to the common relational data model and is highly compatible with the extensible indexing frameworks of existing object-relational database systems allowing the user to define application-specific access methods.

INTRODUCTION

Users of database systems want to manage data of very different types, depending on the particular application area. While office applications, for example, mainly perform simple access and update operations on records of simple data types, spatial data usually have a complex structure and demand specialized operations. It is not a choice for vendors of database management systems to provide data types and management functions for each conceivable domain. So the design of extensible architectures allowing users to adapt systems to their special needs represents an important area in database research.

Traditional relational database systems support only a very limited way of extensibility. All data have to be mapped on rows of flat tables consisting of attributes with simple types like numbers, character strings or dates. For the retrieval and manipulation of data, there exist only generic operations for selecting, inserting, updating and deleting (parts of) rows within tables. Data of more complex types cannot be stored directly as a unit in the database but have to be split across several tables. To restore the data from the system, complex queries with many joins have to be performed. Alternatively, the data can be coded within a large
object which prevents direct access to single components of the data using the database language. Operations on complex types have to be implemented within the application and cannot be used within the database language directly.

Object-oriented database management systems (OODBMSs) seem to provide solutions for most of the cited problems of relational databases. An OODBMS has an extensible type system which allows the user to define new data types (by the nested application of type constructors) together with corresponding operations. The resulting object types then describe the structure as well as the behaviour of the objects based on this type. Furthermore, subtypes (inheriting the properties of their supertypes) can be derived of existing object types.

In order to provide object-oriented and extensibility features also in relational systems, database researchers and manufacturers proposed and implemented corresponding enhancements for the relational model during the last years. The resulting object-relational database management systems (ORDBMSs) retain all features of the relational model, especially the storage of data within tables and the powerful declarative query processing with the relational database language SQL. Beyond that, the object-relational data model introduces abstract data types into relational database servers. Thereby, object-relational database systems may be used as a natural basis to design an integrated user-defined database solution. The ORDBMSs already support major aspects of the declarative embedding of user-defined data types and predicates. In order to achieve a seamless integration of custom object types and predicates within the declarative data definition language (DDL) and data manipulation language (DML), ORDBMSs provide the database developer with extensibility interfaces. They enable the declarative embedding of abstract data types within the built-in optimizer and query processor.

In the following, we categorize possible approaches to incorporate third-party spatial indexing structures into a relational database system by what we call Relational Indexing. After an introduction to ORDBMSs and their extensible indexing facilities, we discuss three different implementations of spatial access methods, including the relational approach, and introduce basic concepts of object-relational spatial access methods. Then the design of the corresponding update and query operations is investigated. Afterwards, we identify two generic schemes for modeling object-relational spatial access methods which are discussed with respect to their support of concurrent transactions and recovery. Finally, some concluding remarks are given.
INDEXING INTERFACES IN OBJECT-RELATIONAL DATABASES

Extensible frameworks are available for most object-relational database systems, including Oracle (Oracle, 1999a; Srinivasan, Murthy, Sundara, Agarwal, & DeFazio, 2000), IBM DB2 (IBM, 1999; Chen et al., 1999), or Informix IDS/UDO (Informix, 1998; Bliujute, Saltenis, Slivinskas, & Jensen, 1999). Custom server components using these built-in services are called data cartridges, database extenders, and data blades, in Oracle, DB2 and Informix, respectively. The open-source ORDBMS PostgreSQL (PostgreSQL, 2002; Stonebraker & Kemnitz, 1991) has the same roots as the commercial database system of Informix and also provides similar extensibility features.

**Declarative Integration.** As an example, we create an object type POLYGON to encapsulate the data and semantics of two-dimensional polygons. Instances of this spatial object type are stored as elements of relational tuples. Figure 1 depicts some of the required object-relational DDL statements in pseudo SQL thus abstracting from technical details which depend on the chosen product. By using the functional binding of the user-defined predicate INTERSECTS, object-relational queries can be expressed in the usual declarative fashion (cf. Figure 2). Provided only with a functional implementation which evaluates the INTERSECTS predicate in

```sql
// Type declaration
CREATE TYPE POINT AS OBJECT (x NUMBER, y NUMBER);
CREATE TYPE POINT_TABLE AS TABLE OF POINT;
CREATE TYPE POLYGON AS OBJECT (points POINT_TABLE,
   MEMBER FUNCTION intersects (p POLYGON) RETURN BOOLEAN);

// Type implementation
// …

// Functional predicate binding
CREATE OPERATOR INTERSECTS (a POLYGON, b POLYGON) RETURN BOOLEAN
BEGIN RETURN a.intersects(b); END;

// Table definition
CREATE TABLE polygons (id NUMBER PRIMARY KEY,
   geom POLYGON);
```

**Figure 1:** Object-relational DDL statements for polygon data
a row by row manner, the built-in optimizer has to include a full-table scan into the execution plan to perform the spatial selection. In consequence, the resulting performance will be very poor for highly selective query regions. As a solution, the extensibility services of the ORDBMS offer a conceptual framework to supplement the functional evaluation of user-defined predicates with index-based lookups.

**Extensible Indexing.** An important requirement for spatial applications is the availability of user-defined access methods. Extensible indexing frameworks proposed by Stonebraker (1986) enable developers to register custom secondary access methods at the database server in addition to the built-in index structures. An object-relational indextype encapsulates stored functions for creating and dropping a custom index and for opening and closing index scans. The row-based processing of selections and update operations follows the iterator pattern (Gamma, Helm, Johnson, & Vlissides, 1995). Thereby, the indextype complements the functional implementation of user-defined predicates. Figure 3 shows some basic indextype methods invoked by extensible indexing frameworks. Additional functions exist to support query optimization, custom joins, and user-defined aggregates. Assuming that we have encapsulated a spatial access method for two-dimensional polygons within the custom indextype SpatialIndex, we may create an index polygons_idx on the geom attribute of the polygons table by submitting the usual DDL statement (cf. Figure 4). If the optimizer decides to include this

```sql
// Region query
SELECT id FROM polygons
WHERE INTERSECTS(geom, :query_region);
```

**Figure 2:** Object-relational region query on polygon data for a region `query_region`

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>index_create(),</td>
<td>Create and drop a custom index.</td>
</tr>
<tr>
<td>index_drop()</td>
<td></td>
</tr>
<tr>
<td>index_open(), index_close()</td>
<td>Open and close a custom index.</td>
</tr>
<tr>
<td>index_fetch()</td>
<td>Fetch the next record from the index that meets the query predicate.</td>
</tr>
<tr>
<td>index_insert(), index_delete(), index_update()</td>
<td>Add, delete, and update a record of the index.</td>
</tr>
</tbody>
</table>

**Figure 3:** Methods for extensible index definition and manipulation
custom index into the execution plan for a declarative DML statement, the appropriate indextype functions are called by the built-in query processor of the database server. Thereby, the maintenance and access of a custom index structure is completely hidden from the user, and the desired data independence is achieved. Furthermore, the framework guarantees any redundant index data to remain consistent with the user data.

Talking to the Optimizer. Query optimization is the process of choosing the most efficient way to execute a declarative DML statement (Yu & Meng, 1998). Object-relational database systems typically support rule-based and cost-based query optimization. The extensible indexing framework comprises interfaces to tell the built-in optimizer about the characteristics of a custom indextype (Stonebraker & Brown, 1999). Figure 5 shows some cost-based functions, which can be implemented to provide the optimizer with feedback on the expected index behavior. The computation of custom statistics is triggered by the usual administrative SQL statements. With a cost model registered at the built-in optimizer framework, the cost-based optimizer is able to rank the potential usage of a custom access method among alternative access paths. Thus, the system supports the generation of efficient execution plans for queries comprising user-defined predicates. This approach preserves the declarative paradigm of SQL, as it requires no manual query rewriting.

<table>
<thead>
<tr>
<th>Function</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>stats_collect(), stats_delete()</td>
<td>Collect and delete persistent statistics on the custom index.</td>
</tr>
<tr>
<td>predicate_sel()</td>
<td>Estimate the selectivity of a user-defined predicate by using the persistent statistics.</td>
</tr>
<tr>
<td>index_cpu_cost(), index_io_cost()</td>
<td>Estimate the CPU and I/O cost required to evaluate a user-defined predicate on the custom index.</td>
</tr>
</tbody>
</table>

Figure 4: Creation of a custom index on polygon data

```
// Index creation
CREATE INDEX polygons_idx ON polygons(geom)
INDEXTYPE IS SpatialIndex;
```

Figure 5: Methods for extensible query optimization
IMPLEMENTATION OF SPATIAL ACCESS METHODS

In the previous section, we have outlined how object-relational database systems support the logical embedding of spatial indextypes into the declarative query language and into the optimizer framework. The required high-level interfaces can be found in any commercial ORDBMS and are continuously improved and extended by the database vendors. Whereas the embedding of a custom indextype is therefore well supported, its actual implementation within a fully-fledged database kernel remains an open problem. In the following, we discuss three basic approaches to implement the low-level functionality of a spatial access method: the integrating, the generic, and the relational approach (cf. Figure 6).

The Integrating Approach

By following the integrating approach, a new spatial access method (AM) is hard-wired into the kernel of an existing database system (cf. Figure 6b). In consequence, the required support of ACID properties, including concurrency control and recovery services (CC&R) has to
be implemented from scratch and linked to the corresponding built-in components. Furthermore, a custom gateway to the built-in storage, buffer, and log managers has to be provided by the developer of the new AM. Most standard primary and secondary storage structures are hard-wired within the database kernel, including plain table storage, hash indexes, bitmap indexes, and B+-trees. Only a few non-standard access methods have been implemented into commercial systems in the same way, including the R-Link-tree in Informix IDS/UDO for spatially extended objects (Informix, 1999) and the UB-tree in TransBase/HC for multidimensional point databases (Ramsak et al., 2000). The integrating approach comprises the Extending Approach and the Enhancing Approach.

The Extending Approach. Adding a completely new access method to a database system is quite expensive because in addition to the actual index algorithms all the concurrency, recovery and page management has to be implemented for the new access method (R-Link-Tree, Bitmaps, External Memory Interval Tree). Carey et al. (1990) guess that the actual index algorithms only comprise about 30 percent of the overall code for the access method while the other 70 percent are needed to integrate the access method properly into the database system.

Several approaches to facilitate the implementation of access methods and other components for special-purpose database systems have been proposed in database research. Under the assumption that it is practically impossible to implement a database management system capable to fulfill the demands of arbitrary application domains, tools and generic database components have been developed that should enable a domain specialist to implement his or her required database system with minimum effort, may it be a document management system or a geographical information system. The resulting systems might have completely different characteristics, e.g. different query languages, different access methods, different storage management, and different transaction mechanisms.

The database system toolkit EXODUS (Carey et al., 1990; Carey et al., 1991) provides a storage manager for objects, a library of access methods, a library of operator methods for the data model to generate, a rule-based generator for query optimizers, tools for constructing query languages and a persistent programming language for the definition of new access methods and query operators. Using these “tools”, the domain specialist can build an application-specific database system with suitable access methods. Another system of this category is the database generator GENESIS (Batory et al., 1990) which provides a set of composable storage and indexing primitives and a “database system compiler” for assembling an appropriate storage manager from a specification.
Unfortunately, these universally extensible database systems have essentially proven to be hard to use in practice. Though these systems support the user with the implementation of single database components, still a lot of expertise is required to use them. In some ways, they are also a bit too inflexible and incomplete to implement a fully-fledged database management system. So in practice, only few databases have been implemented using such toolkits or generators.

**The Enhancing Approach.** In contrast to the extending approach the enhancing approach is much cheaper, since already existing access methods are augmented to support a broader range of data. The code of the access methods to be enhanced has to be adapted in such a way that it gets independent of the indexed data type. As an example, Figure 7a depicts the pseudocode of a B-tree index routine \textit{next\_node} that determines (for a given node and a key value) the node that has to be visited next within a B-tree traversal. This routine only works on key values of type \textit{integer}, i. e. an index using this routine can only be created on columns

\begin{verbatim}
next_node(n:node; key:integer);
    ...
    while n.son[index].value < key
        increment(index);
        next := fetch(n.son[index].address);
    ...
end;
\end{verbatim}

\textbf{a) next\_node routine handling keys of type integer}

\begin{verbatim}
next_node(n:node; key:real);
    ...
    while n.son[index].value < key
        increment(index);
        next := fetch(n.son[index].address);
    ...
end;
\end{verbatim}

\textbf{b) next\_node routine handling keys of type real}

\begin{verbatim}
template <keytype>
next_node(n:node; key:keytype);
    ...
    while lessthan(n.son[index].value, key)
        increment(index);
        next := fetch(n.son[index].address);
    ...
end;
\end{verbatim}

\textbf{c) next\_node routine handling arbitrary data types}

**Figure 7:** B-tree routine next\_node for different data types
of type `integer`. In order to create B-tree indexes also on columns of type `real`, the type of the key parameter has to be changed accordingly (Figure 7b). In general, to support arbitrary (ordered) types, the B-tree code has to be modified in such a way that it can handle key parameters of any type. Figure 7c depicts a type-independent version of the `next_node` routine. Here, the key type is not determined but has to be instantiated when applying the index. The function `lessthan` has the same functionality as the operator ‘<’ for built-in types. If the user defines a new type and wants to use the enhanced B-tree index for columns of this type, he or she has to provide a corresponding `lessthan` function that can handle values of the new type. Alternatively, the built-in operator ‘<’ could be overloaded, if the database system used supports this. As a further example, if the user defines a new type `FracNum` for the storage of fraction numbers (consisting of numerator and denominator) in the database system (Figure 8a), he or she has to implement a special version of the function `lessthan` that takes two fraction numbers as parameters (Figure 8b). Whenever the routine `next_node` is called with a key parameter of type `FracNum`, the newly defined version of `lessthan` is used.

In general, to enhance (generalize) an access method in this way, all type-specific operations within the code of the access method have to be identified and isolated so that the user can provide overloaded versions of these operations for his user-defined types. It is necessary to note that not every access method is appropriate for every data type. B-trees e. g. only can be used for types with a linear ordering. In contrast, R-trees are designed to support access to spatially extended and to multi-dimensional data. Depending on the access method and the predicates to be supported by the index, the user has to implement corresponding operations for new data types. To use an enhanced B-tree index, the user must provide implementations

```sql
CREATE TYPE FracNum (num INTEGER; denom INTEGER)

CREATE FUNCTION lessthan (f1 FracNum, f2 FracNum)
RETURN BOOLEAN
LANGUAGE SQL DETERMINISTIC
BEGIN
    RETURN (f1.num/f1.denom) < (f2.num/f2.denom);
END

b) Function lessthan for comparing fraction numbers

Figure 8: User-defined data type FracNum
```
of the usual comparison operators ‘<’, ‘<’, ‘>’, ‘>’, and ‘=’ for a new data type whereas an R-tree index requires spatial operations like “overlaps”, “contains”, “within”, or “equals”.

A further possibility to enhance existing access methods is to implement functional indexes that give quick access to the results of a function defined on the attributes of a table. The type of the function value has to be supported by the enhanced index.

In conclusion, we identify the following properties of the integrating approach:

**Implementation:** The implementation of a new spatial AM becomes very sophisticated and tedious if writing transactions have to be supported (Brown, 2001). In addition, the code maintenance is a very complex task, as new kernel functionality has to be implemented for any built-in access method. Moreover, the tight integration within the existing kernel source produces a highly platform-dependent solution tailor-made for a specific ORDBMS. The enhancement of pre-existing access methods to support user-defined data types and functional indexes is a straightforward task but does not really augment the functionality of the database server (in the sense of having new ways for query processing).

**Performance:** The integrating approach potentially delivers the maximal possible performance, if the access method is implemented in a closed environment, and the number of context switches to other components of the database kernel is minimized.

**Availability:** The implementation requires low-level access to most kernel components. If the target ORDBMS is not distributed as open-source, the affected code and documentation will not be accessible to external database developers.

To sum up, the integrating approach is the method of choice only for a few, well-selected access methods serving the requirements of general database applications. It is not feasible for the implementation of rather specialized access methods.

**The Generic Approach**

To overcome the restrictions of the integrating method, Hellerstein, Naughton and Pfeffer (1995) proposed a generic approach to implement new access methods in an ORDBMS. Their *Generalized Search Tree (GiST)* has to be built only once into an existing database kernel. The GiST serves as a high-level framework to plug in block-based tree structures with full ACID support (cf. Figure 6c).

As in the previous approaches, the database implementor has to integrate the extensibility framework into the database server regarding all tedious tasks like concurrency and recovery. Once implemented, a domain specialist can use the GiST framework to derive new index
types for particular applications. In contrast to database toolkits or generators, the index implementor does not have to stop the database server and recompile it every time an index type is added. It is just necessary to implement (overload) a number of predefined functions which define the behaviour of keys in the tree. This is quite similar as in the enhanced index approach at first glance. However, while enhanced indexes only support new data types for already existing index structures, it is possible to support completely new query predicates with the GiST framework. B-trees and R-trees are both derivable from the GiST framework, for example. Such derived index types may not be as performant as directly integrated ones but require much less effort to realize.

Many extensions to the GiST framework have been presented, including generic support for concurrency and recovery (Kornacker, Mohan, & Hellerstein, 1997), and additional interfaces for nearest-neighbor search, ranking, aggregation, and selectivity estimation (Aoki, 1998). In detail, the GiST approach has the following characteristics:

**Implementation:** Whereas the implementation of block-based spatial access methods on top of the GiST framework can be done rather easily, the intruding integration of the framework itself remains a very complex task. As an advantage, an access method developed for GiST can basically be employed on any ORDBMS that supports this framework. In contrast to the generic GiST implementation, the specialized functionality of a new access method is therefore platform independent.

**Performance:** Although the framework induces some overhead, we can still achieve high performance for GiST-based access methods. Kornacker (1999) has shown that they may even outperform built-in index structures by minimizing calls to user-defined functions.

**Availability:** Due to its complex implementation, the GiST framework is not generally available in present-day systems. To our best knowledge, it has only been implemented in the open-source system PostgreSQL but without concurrent access and write-ahead logging of updates for derived indexes. It is an open question, whether and when a comparable functionality with industrial-strength implementation will be a standard component of major commercial ORDBMSs.

The GiST concept basically delivers the desired properties to implement spatial access methods. It delegates crucial parts of the implementation to the database vendors. Unfortunately, its full functionality is not available in any major commercial database system at present. Furthermore, database extensions should generically support many database platforms.
Thus, the GiST concept would have to be implemented not only for one, but for all major ORDBMSs.

The Relational Approach

A natural way to avoid the above obstacles is to map the spatial index structure to a relational schema organized by built-in access methods (cf. Figure 6d). Such relational access methods are designed to operate on top of a relational query language. They require no extension or modification of the database kernel, and, thus, any off-the-shelf ORDBMS can be employed as it is. We identify the following advantages for the relational approach:

Implementation: As no internal modification or extension to the database server is required, a relational access method can be implemented and maintained with less effort. Substantial parts of the custom access semantics may be expressed by using the declarative DML. Thereby, the implementation exploits the existing functionality of the underlying ORDBMS rather than duplicating basic database services as done in the integrating and generic approaches. Moreover, if we use a standardized DDL and DML like SQL:1999 (ANSI, 1999) to implement the low-level interface of our access method, the resulting code will be platform independent.

Performance: The major challenge in designing a relational access method is to achieve both a high usability and performance. The capability and efficiency of the relational approach was proven for interval data (Kriegel, Pötte, & Seidl, 2000; Kriegel, Pfeifle, Pötte, & Seidl, 2002) and 2D/3D spatial data (Kriegel, Pötte, & Seidl, 2001; Kriegel, Müller, Pötte, & Seidl, 2001).

Availability: By design, a relational access method is supported by any relational database system. It requires the same functionality as an ordinary database user or a relational database application.

By following the relational approach to implement spatial access methods, we obtain a natural distinction between the basic services of all-purpose database systems and specialized, application-specific extensions. By restricting database accesses to the common SQL interface, spatial access methods and query procedures are well-defined on top of the core server components. In addition, a relational access method immediately benefits from any improvement of the ORDBMS infrastructure.
The basic idea of relational access methods relies on the exploitation of the built-in functionality of existing database systems. Rather than extending any internal component of the database kernel, a relational access method just uses the native data definition and data manipulation language to process updates and queries on abstract data types. Without loss of generality, we assume that the underlying database system implements the standardized Structured Query Language SQL-92 (ANSI, 1992) with common object-relational enhancements in the sense of SQL:1999 (ANSI, 1999), including object types and collections.

Paradigms of Access Methods

A relational access method delegates the management of persistent data to an underlying relational database system by strictly implementing the index definition and manipulation on top of an SQL interface. Thereby, the SQL layer of the ORDBMS is employed as a virtual machine managing persistent data. Its robust and powerful abstraction from block-based secondary storage to the object-relational model can then be fully exploited. This concept also perfectly supports database appliances, i.e. dedicated database machines running the ORDBMS as a specialized operating system (Keim & Prawirohardjo, 1992; Oracle, 2000). We add the class of relational access methods as a third paradigm to the known paradigms of access methods for database management systems:

Main Memory Access Methods (Figure 9a). Typical applications of these techniques can be found in main memory databases (DeWitt et al., 1985; Garcia-Molina & Salem, 1992) and in

**Figure 9**: Paradigms and characteristics of access methods:

a) main memory access methods,  
b) block-oriented access methods,  
c) relational access methods
the field of computational geometry (Preparata & Shamos, 1993). A popular example taken from the latter is the binary Interval Tree (Edelsbrunner, 1983). It serves as a basic data structure for plane-sweep algorithms, e.g. to process intersection joins on rectangle sets. Main memory structures are not qualified for indexing persistent data, as they disregard the block-oriented access to secondary storage.

**Block Oriented Access Methods** (Figure 9b). These structures are designed to efficiently support the block-oriented I/O from and to external storage and are well suited to manage large amounts of persistent data. The External Memory Interval Tree (Arge & Vitter, 1996) is an example for the optimal externalization of a main memory access method. Its analytic optimality is achieved by adapting the fanout of the Interval Tree to the disk block size. In the absence of a generalized search tree framework (Hellerstein et al., 1995), the implementation of such specialized storage structures into existing database systems, along with custom concurrency control and recovery services, is very complex, and furthermore, requires intrusive modifications of the database kernel (Ramsak et al., 2000).

**Relational Access Methods** (Figure 9c). In contrast, relational access methods including the Relational Interval Tree (Kriegel et al., 2000) are designed to operate on relations rather than on dedicated disk blocks. The persistent storage and block-oriented management of the relations is delegated to the underlying database server. Therefore, the robust functionality of the database kernel including concurrent transactions and recovery can potentially be reused. A primary clustering index can be achieved by also delegating the clustering to the ORDBMS. For this, the payload data has to be included into the index relations and the clustering has to be enabled by organizing these tables in a cluster or as index-organized tables (Srinivasan, Das, et al., 2000).

**Relational Storage of Index Data**

In the following, we will discuss the basic properties of relational access methods with respect to the storage of index data, query processing and the overhead for transaction semantics, concurrency control, and recovery services. We start with a basic definition:

**Definition 1 (Relational Access Method).**

An access method is called a relational access method, iff any index-related data is exclusively stored in and retrieved from relational tables. An instance of a relational access method
is called a relational index. The following tables comprise the persistent data of a relational index:

(i) User table: a single table, storing the original user data being indexed.

(ii) Index tables: \( n \) tables, \( n \geq 0 \), storing index data derived from the user table.

(iii) Meta table: a single table for each database and each relational access method, storing \( O(1) \) rows for each instance of an index.

The stored data is called user data, index data, and meta data.

To illustrate the concept of relational access methods, Figure 10 presents the minimum bounding rectangle list (MBR-List), a very simple example for indexing two-dimensional polygons. The user table is given by the object-relational table polygons (Figure 10a), comprising attributes for the polygon data type (geom) and the object identifier (id). Any spatial query can already be evaluated by sequentially scanning this user table. In order to speed up spatial selections, we decide to define an MBR-List polygons_idx on the user table. Thereby, an index table is created and populated (Figure 10b), assigning the minimum bounding rectangles (mbr) of each polygon to the foreign key id. Thus, the index table stores information purely derived from the user table. All schema objects belonging to the relational index, in particular the name of the index table, and other index parameters are stored in a global meta table (Figure 10c).

In order to support queries on the index tables, a relational access method can employ any built-in secondary indexes, including hash indexes, B+-trees, and bitmap indexes. Alternatively, index tables may be clustered by appropriate primary indexes. Consequently, the relational access method and the database system cooperate to maintain and retrieve the in-
dex data (DeFazio, Daoud, Smith, & Srinivasan, 1995). This basic approach of relational index- ing has already been applied in many existing solutions, including Linear Quadtrees (Tropf & Herzog, 1981; Ravada & Sharma, 1999; Freytag, Flasza, & Stillger, 2000) and Relational R-trees (Ravi, Ravada, Sharma, & Banerjee, 1999) for spatial databases, Relational X-trees (Berchtold, Böhm, Kriegel, & Michel, 1999) for high- dimensional nearest-neighbor search, or inverted indexes for information retrieval on text documents (DeFazio et al., 1995).

OPERATIONS ON RELATIONAL ACCESS METHODS

In the strict sense of the Definition, the procedural code of an arbitrary block-oriented storage structure can immediately be transformed to a relational access method by replacing each invocation of the underlying block manager by an SQL-based DML operation. Thus, the original procedural style of an index operation remains unchanged, whereas its I/O requests are now executed by a fully-fledged RDBMS. The object-relational database server is thereby reduced to a plain block manager. In consequence, only a fraction of the existing functionality of the underlying database server is exploited. In this section, we define operations on relational access methods which maximize the architecture-awareness postulated in (Jensen & Snodgrass, 1999). This can be achieved by using declarative operations.

Cursor-Bound Operations

In order to guarantee a better exploitation of the database infrastructure, we have to restrict the possible number of DML operations submitted from a procedural environment:

**Definition 2 (Cursor-Bound Operation).** A query or update operation on a relational access method is termed cursor-bound, iff the corresponding I/O requests on the index data can be performed by submitting $O(1)$ DML statements, i.e. by sequentially and concurrently opening in total $O(1)$ cursors provided by the underlying RDBMS.

Cursor-bound operations on relational access methods are largely bound to the declarative DML engine of the underlying RDBMS rather than to user-defined opaque code. Thus, the database server gains the responsibility for significant parts of the query and update semantics. Advantages of this approach include:

- **Declarative Semantics.** Large parts of a cursor-bound operation are expressed by using declarative SQL. By minimizing the procedural part and maximizing the declarative part
of an operation, the formal verification of the semantics is simplified if we can rely on
the given implementation of SQL to be sound and complete.

- **Query Optimization.** Whereas the database engine optimizes the execution of single,
closed-form DML statements, a joint execution of multiple, independently submitted
queries is very difficult to achieve (Sellis, 1988; Chen & Dunham, 1998; Braunmüller,
Ester, Kriegel, & Sander, 2000). By using only a constant number of cursors, the
RDBMS captures significant parts of the operational semantics at once. In particular,
complex I/O operations including external sorting, duplicate elimination or grouping
should be processed by the database engine, and not by a user-defined procedure.

- **Cursor Minimization.** The CPU cost of opening a variable number of cursors may be-
come very high. For typical applications, the resulting overhead sums up to 30% of the
total processing time (Ramsak et al., 2000). In some experiments, we even reached bar-
rrier crossing cost of up to 75% for submitting a variable number of pre-parsed DML
statements out of a stored procedure. For cursor-bound operations, the relatively high
cost of opening and fetching multiple database cursors remains constant with respect to
the complexity of the operation and the database size.

**Cursor-Driven Operations**

A very interesting case occurs if the potential result of a cursor-bound operation can be
retrieved as the immediate output of a *single* cursor provided by the DBMS. Thus, the sem-
antics is revealed to the database server at once in its full completeness:

**Definition 3 (Cursor-Driven Operation).** A cursor-bound operation on a relational access
method is called *cursor-driven*, iff it can be divided into two consecutive phases:

(i) *Procedural phase:* In the first phase, index parameters are read from the meta tables.
Query specifications are retrieved and data structures required for the actual query execution
may be prepared by user-defined procedures and functions. Additional DML operations on
user data or index data are not permitted.

(ii) *Declarative phase:* In the second phase, only a single DML statement is submitted to
the ORDBMS, yielding a cursor on the final results of the index scan which requires no post-
processing by user-defined procedures or functions.

Note that any cursor-driven operation is also cursor-bound, while all I/O requests on the
index data are driven by a single declarative DML statement. The major advantage of cursor-
driven operations is their smart integration into larger execution plans. After the completion
of the procedural phase, the single DML statement can be executed with arbitrary groupings and aggregations, supplemented with additional predicates, or serve as a row source for joins. Furthermore, the integration into extensible indexing frameworks is facilitated, as the cursor opened in the declarative phase can be simply pipelined to the index scan routine. Note that the ability to implement cursor-bound and cursor-driven operations heavily relies on the expressive power of the underlying SQL interface, including the availability of recursive queries (Libkin, 2001).

The single DML statement submitted in the declarative phase may contain user-defined functions. The CPU cost of cursor-driven operations is significantly reduced, if the number of barrier crossings due to calls to user-defined functions is minimized (Kornacker, 1999). We can achieve this by preprocessing any required transformation, e.g. of a query specification, in the procedural phase and by bulk-binding the prepared data to the query statement with the help of transient collections. If such data structures become very large, a trade-off has to be achieved between the minimization of barrier crossings and the main-memory footprint of concurrent sessions. Splitting a single query into multiple cursor-driven operations can then be beneficial.

To pick up the MBR-List example of the previous section, Figure 11a shows a simple window query on the database of two-dimensional polygons, testing the exact geometry of each stored polygon for intersection with the query rectangle. In order to use the relational index as primary filter, the query has to be rewritten into the form of Figure 11b. An efficient execution plan for the rewritten query may first check the intersection with the stored bounding boxes, and refine the result by performing the equijoin with the polygons table. Note that the window query is a cursor-driven operation on the MBR-List, having an empty procedural phase. Therefore, the index-supported query can be easily embedded into a larger context as shown in Figure 11c. Already this small example shows that an object-relational wrapping of relational access methods is essential to control redundant data in the index tables and to avoid manual query rewriting. The usage of an extensible indexing framework preserves the physical independence of DML operations and enables the usual query optimization.
Although similarity queries or nearest neighbor queries ("return the k polygons closest to a query point wrt. to a given metric") can also be performed in a cursor-driven way by using the order-by clause together with a top-k-filter, the efficiency of this approach is rather questionable (Carey & Kossmann, 1997).

**Generic Schemes for Object-Relational Spatial Indexing**

As an immediate result of the relational storage of index data and meta data, a relational index is subject to the built-in transaction semantics, concurrency control, and recovery services of the underlying database system. In this section, we discuss the effectiveness and performance provided by the built-in services of the ORDBMS on relational access methods. For that purpose, we identify two generic schemes for the relational storage of index data, the navigational scheme and the direct scheme.

**Navigational Scheme of Index Tables**

**Definition 4 (Navigational Scheme).**

Let \( P = (T, R_1, ..., R_n) \) be a relational access method on a data scheme \( T \) and index schemes \( R_1, ..., R_n \). We call \( P \) navigational \( \iff (\exists t \subseteq T) (\exists r_i \subseteq R_i, 1 \leq i \leq n): \) at least one \( \rho \in r_i \) is associated with rows \( \{\tau_1, ..., \tau_m\} \subseteq t \) and \( m > 1 \).

```
SELECT id FROM polygons
WHERE geom INTERSECTS BOX((0,0),(100,100));

a) Window query on the user table.

SELECT usr.id AS id FROM polygons usr, polygons_mbr idx
WHERE idx.mbr INTERSECTS BOX((0,0),(100,100))
AND idx.id = usr.id
AND usr.geom INTERSECTS BOX ((0,0),(100,100));

b) Window query using the relational index as primary filter.

SELECT id FROM polygon_type
WHERE type = 'LAKE'
AND id IN (  SELECT usr.id FROM polygons usr, polygons_mbr idx
WHERE idx.mbr INTERSECTS BOX((0,0),(100,100))
AND idx.id = usr.id
AND usr.geom INTERSECTS BOX ((0,0),(100,100))
);  
c) Index-supported window subquery.
```

![Figure 11: Window queries on two-dimensional polygons](image)

Although similarity queries or nearest neighbor queries ("return the k polygons closest to a query point wrt. to a given metric") can also be performed in a cursor-driven way by using the order-by clause together with a top-k-filter, the efficiency of this approach is rather questionable (Carey & Kossmann, 1997).
Therefore, a row in an index table of a navigational index may logically represent many objects stored in the user table. This is typically the case for hierarchical structures that are mapped to a relational schema. Consequently, an index table contains data that is recursively traversed at query time in order to determine the resulting tuples. Examples for the navigational scheme include the Oracle Spatial R-tree (Ravi et al., 1999) and the Relational X-tree (Berchtold et al., 1999) which store the nodes of a tree directory in a flat table. To implement a navigational query as a cursor-bound operation, a recursive version of SQL like SQL:1999 (ANSI, 1999; Eisenberg & Melton, 1999) is required.

Although the navigational scheme offers a straightforward way to simulate any hierarchical index structure on top of a relational data model, it suffers from the fact that navigational data is locked like user data. As two-phase locking on index tables is too restrictive, the possible level of concurrency is unnecessarily decreased. For example, uncommitted node splits in a hierarchical directory may lock entire subtrees against concurrent updates. Built-in indexes solve this problem by committing structural modifications separately from content changes (Kornacker & Banks, 1995). Unfortunately, this approach is not feasible on the SQL layer without braking up the user transaction. A similar overhead exists with logging, as atomic actions on navigational data, e.g. node splits, are not required to be rolled back in order to keep the index tables consistent with the data table. Therefore, relational access methods implementing the navigational scheme are only well suited for read-only or single-user environments.

Relational R-trees – A Spatial Example for the Navigational Scheme

We illustrate the properties and drawbacks of the navigational scheme by the example of Relational R-trees, like they have been used by the Oracle developers Ravi Kanth et al. (1999). Figure 12 depicts a hierarchical R-tree along with a possible relational mapping (\(page\_id, page\_lev, son\_id, son\_mbr\)). The column \(page\_id\) contains the logical page identifier, while \(page\_lev\) denotes its level in the tree. Thereby, 0 marks the level of the data objects, and 1 marks the leaf level of the directory. The attribute \(son\_id\) contains the \(page\_id\) of the connected entry, while \(son\_mbr\) stores its minimum bounding rectangle. Thus, \(page\_id\) and \(son\_id\) together comprise the primary key. In our example, the logical page 2 represents a partition of the data space which contains the polygons \(A\) and \(B\). The corresponding index row \((1, 2, 2, \ldots)\) is therefore logically associated with the rows \((A, \ldots)\) and \((B, \ldots)\) in the
polygons user table (cf. Figure 10). Thus, the Relational R-tree implements the navigational scheme of relational access methods.

The severe overhead of the navigational scheme already becomes obvious if a transaction inserts a new polygon, and subsequently enlarges the bounding box of a node, e.g. of the root node. Due to the common two-phase locking, this transaction will hold an exclusive lock on the row (ROOT, 3, 1, …) until commit or rollback. During this time, no concurrent transaction can insert polygons that induce an enlargement of the root region. The database server has to guarantee non-blocking reads (Oracle, 1999b) to support at least concurrent queries on the Relational R-tree index. If the low concurrency of the Relational R-tree is acceptable, the relational mapping opens up a wide range of potential improvements (Kriegel, Pfeifle, Pötte, & Seidl, 2003).

To support the navigation through the R-tree table at query time, a built-in index can be created on the page_id column. Alternatively, the schema can be transformed to NF² (non-first normal form), where page_id alone represents the primary key, and a collection of (son_id, son_mbr) pairs is stored with each row. In this case, the static storage location of each tuple can be used as page_id, avoiding the necessity of a built-in index. A cursor-driven primary filter for a window query using recursive SQL is shown in Figure 13. We expect that future implementations of the SQL:1999 statement yield a depth-first traversal which is already hard-wired into the existing CONNECT BY clause of the Oracle server. The effectiveness of cursor-driven operations is illustrated by the fact that the depicted statements already comprise the complete, pipelined query processing on the R-tree index.

Figure 12: Relational mapping of an R-tree directory
Definition 5 (Direct Scheme).

Let $P = (T, R_1, \ldots, R_n)$ be a relational access method on a data scheme $T$ and index schemes $R_1, \ldots, R_n$. We call $P$ direct $\iff$ $(\forall t \subseteq T) (\forall r_i \subseteq R_i, 1 \leq i \leq n)$: each $\rho \in r_i$ is associated with a single row $\tau \in t$.

In consequence, for a relational access method of the direct scheme, each row in the user table is directly mapped to a set of rows in the index tables. Inversely, each row in an index table exclusively belongs to a single row in the user table. In order to support queries, the index table is organized by a built-in index, e.g. a B+-tree. Examples for the direct scheme include our MBR-List (cf. Figure 10), the Linear Quadtree (Samet, 1990), the one-dimensional Relational Interval Tree (Kriegel et al., 2000) and its optimization for interval sequences and multidimensional queries (Kriegel, Pötke, et al., 2001).

The drawbacks of the navigational scheme with respect to concurrency control and recovery are not shared by the direct scheme, as row-based locking and logging on the index tables can be performed on the granularity of single rows in the user tables. For example, an update of a single row $r$ in the user table requires only the synchronization of index rows ex-
clusively assigned to \( r \). As the acquired locks are restricted to \( r \) and its exclusive entries in the index tables, they do not unnecessarily block concurrent operations on other user rows. In contrast to navigational indexes, the direct scheme inherits the high concurrency and efficient recovery of built-in tables and indexes.

**Relational Interval Trees – An Example for the Direct Scheme**

An access method implementing the direct scheme is the Relational Interval Tree (Kriegel et al., 2000; Kriegel, Pötke, et al., 2001). Being a relational storage structure for interval data \((\text{lower}, \text{upper})\), it follows the concept of Edelsbrunner’s main-memory interval tree (Edelsbrunner, 1983; Preparata & Shamos, 1993) by design and guarantees the optimal complexity for storage space and for I/O operations when updating or querying large sets of intervals.

The structure of an RI-tree consists of a binary tree of height \( h \) which covers the range \([1, 2^h - 1]\) of potential interval bounds. It is called the virtual backbone of the RI-tree since it is not materialized but only the root value \( 2^{h-1} \) is stored persistently in a metadata table. Traversals of the virtual backbone are performed purely arithmetically by starting at the root value and proceeding in positive or negative steps of decreasing length \( 2^{h-i} \), thus reaching any desired value of the data space in \( O(h) \) CPU time and without causing any I/O operation. For the relational storage of intervals, the node values of the tree are used as artificial keys: Upon insertion of an interval, the first node that hits the interval when descending the tree from the root node down to the interval location is assigned to that interval.

An instance of the RI-tree then consists of two relational indexes which in an extensible indexing environment are preferably managed as index-organized tables. The indexes obey the relational schema \( \text{lowerIndex} (\text{node}, \text{lower}, \text{id}) \) and \( \text{upperIndex} (\text{node}, \text{upper}, \text{id}) \) and store the artificial key value \( \text{node} \), the bounds \( \text{lower} \) and \( \text{upper} \), and the \( \text{id} \) of each interval. An interval is represented by exactly one entry in each of the two indexes, and for inserting or deleting intervals, the \( \text{node} \) values are determined arithmetically without any I/O operation.
The illustration in Figure 14 provides an example for the RI-tree. Let us assume the intervals (2,13) for Mary, (4,23) for John, (10,21) for Bob, and (21,30) for Ann (Fig. 14a). The virtual backbone is rooted at 16 and covers the data space from 1 to 31 (Fig. 14b). The intervals are registered at the nodes 8, 16, and 24. The interval (2,13) for Mary is represented by the entries (8, 2, Mary) in the lowerIndex and (8, 13, Mary) in the upperIndex since 8 is the registration node, and 2 and 13 are the lower and upper bound, respectively (Fig. 14c).

Again to minimize barrier crossings between the procedural runtime environment and the declarative SQL layer, an interval intersection query \((\text{lower}, \text{upper})\) is processed in two steps. The procedural query preparation step descends the virtual backbone from the root node down to \(\text{lower}\) and to \(\text{upper}\), respectively. The traversal is performed arithmetically without causing any I/O operations, and the visited nodes are collected in two main-memory tables, \(\text{left queries}\) and \(\text{right queries}\), as follows: nodes to the left of \(\text{lower}\) may contain intervals which overlap \(\text{lower}\) and are inserted into \(\text{left queries}\). Analogously, nodes to the right of \(\text{upper}\) may contain intervals which overlap \(\text{upper}\) and are inserted into \(\text{right queries}\). Whereas these nodes are taken from the paths, the set of all nodes between \(\text{lower}\) and \(\text{upper}\) belongs to the so-called \(\text{inner query}\) which is represented by a single range query on the node values. All

**Figure 14**: Example for an RI-tree.
intervals registered at nodes from the *inner query* are guaranteed to intersect the query and, therefore, will be reported without any further comparison. The query preparation step is purely based on main memory and requires no I/O operations.

In the subsequent declarative query processing step, the transient tables are joined with the relational indexes `upperIndex` and `lowerIndex` by a single, three-fold SQL statement (Fig. 15). The upper bound of each interval registered at nodes in *left queries* is compared to `lower`, and the lower bounds of intervals stemming from *right queries* are compared to `upper`. The *inner query* corresponds to a simple range scan over the intervals with nodes in `(lower, upper)`.

Recently, a further relational access method implementing the direct scheme was presented (Arge & Chatham, 2003). Like the Relational Interval Tree, the *Relational Priority Search Tree* is a storage structure for handling interval data.

Kriegel et al. (2003) describe the *Linear Quadtree*, another access method with direct scheme implementation.

**CONCLUSIONS**

We presented the concept of object-relational spatial access methods which employ the infrastructure and functionality of existing object-relational database systems to provide efficient execution plans for the evaluation of user-defined predicates. We introduced cursor-bound and cursor-driven operations to maximize the achievable declarativity, usability and performance of operations. We identified two generic schemes for the relational mapping of index data, each having different properties with respect to the built-in locking and logging mechanisms of the underlying database engine: Whereas the *navigational scheme* seems only appropriate for single-user or read-only databases, the *direct scheme* fully preserves the effectivity and efficiency of built-in transactions, concurrency control, and recovery services. The presented concepts have been illustrated by four spatial examples: The *MBR-List*, a

```sql
SELECT id FROM upperIndex i, :leftQueries q
  WHERE i.node = q.node AND i.upper >= :lower
UNION ALL
SELECT id FROM lowerIndex i, :rightQueries q
  WHERE i.node = q.node AND i.lower <= :upper
UNION ALL
SELECT id FROM lowerIndex // or upperIndex
  WHERE node BETWEEN :lower AND :upper;
```

**Figure 15:** SQL statement for an intersection query
trivial relational access method for demonstration purposes, the *Relational R-tree* showing the navigational scheme, along with *Relational Interval Tree*, an access method implementing the direct scheme.

Future research may investigate whether there are generic patterns to develop a relational indexing scheme for any given index structure. Again, a careful analysis of the potentials and the overhead of relational data management is a major point of interest. The development of more powerful extensibility frameworks supporting features as generic indexing interfaces and user-defined join algorithms is a challenge for forthcoming years.

**REFERENCES**


