The Paradigm of Relational Indexing: A Survey

Hans-Peter Kriegel[1], Martin Pfeifle [1], Marco Pötke[2] and Thomas Seidl[3]
[1]University of Munich, {kriegel, pfeifle}@dbs.informatik.uni-muenchen.de
[2]sd&m AG software design & management, marco.poetke@sdm.de
[3]Aachen University (RWTH), seidl@informatik.rwth-aachen.de

Abstract: In order to achieve efficient execution plans for queries comprising user-defined data types and predicates, the database system has to be provided with appropriate index structures, query processing methods, and optimization rules. Although available extensible indexing frameworks provide a gateway to seamlessly integrate user-defined access methods into the standard process of query optimization and execution, they do not facilitate the actual implementation of the 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 in this paper the paradigm of relational 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.

1 Introduction

The design of extensible architectures represents an important area in database research. The object-relational data model marked an evolutionary milestone by introducing 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 DDL and 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. Corresponding frameworks are available for most object-relational database systems, including Oracle [Ora 99a] [SMS+00], IBM DB2 [IBM99] [CCF+99], or Informix IDS/UDO [Inf98] [BSSJ99]. Custom server components using these built-in services are called *data cartridges*, *database extenders,* and *data blades*, in Oracle, DB2 and Informix, respectively.

In this paper, we categorize possible approaches to incorporate third-party indexing structures into a relational database system what we call *Relational Indexing*. Following this introduction about ORDBMS and their extensible indexing facilities, Section 2 discusses three different implementations of user-defined access methods, including the relational approach. In Section 3, basic concepts of relational access methods are introduced, and in Section 4, the design of the corresponding update and query operations are investi-

```
// 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

gated. In Section 5, we identify two generic schemes for modeling relational access methods which are discussed with respect to their support of concurrent transactions and recovery. The paper is concluded in Section 6.

Declarative Integration. As an example, we create an object type *POLYGON* to encapsulate the data and semantics of two-dimensional polygons. Instances of this custom 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 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.

```
// 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*

Function	Task
index_create(), index_drop()	Create and drop a custom index.
index_open(), index_close()	Open and close a custom index.
index_fetch()	Fetch the next record from the index that meets the query predicate.
index_insert(), index_delete(), index_update()	Add, delete, and update a record of the index.

Figure 3: Methods for extensible index definition and manipulation

Extensible Indexing. An important requirement for applications is the availability of user-defined access methods. Extensible indexing frameworks proposed by Stonebraker [Sto86] 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 [GHJV95]. 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 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. 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. Figure 5 shows some cost-based functions, which can be implemented to provide the optimizer with feedback on the expected index behavior. The computation

```
// Index creation

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

Figure 4: Creation of a custom index on polygon data

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

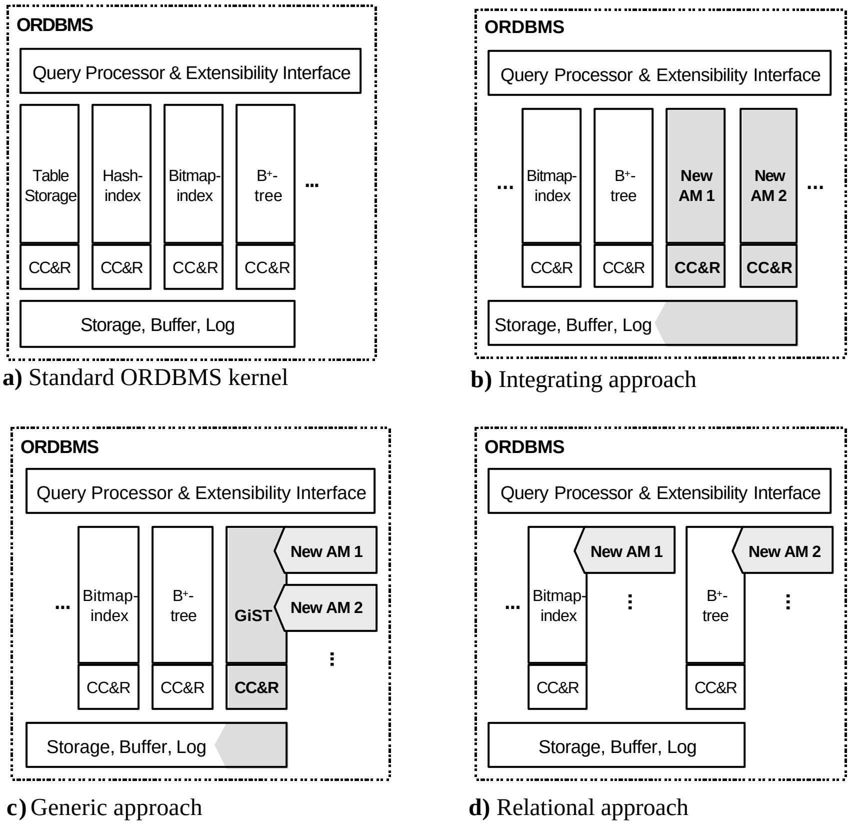
Figure 5: Methods for extensible query optimization

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.

2 Implementation of Access Methods

In the previous section, we have outlined how object-relational database systems support the logical embedding of custom 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 user-defined access method: the *integrating*, the *generic*, and the *relational approach* (cf. Figure 6).

Integrating Approach. By following the integrating approach, a new 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 [Inf99] and the *UB-tree* in TransBase/HC for multidimensional point databases [RMF+00]. The integrating approach comprises the *Extending Approach* and the *Enhancing Approach.* The extending approach is expensive, since it is really adding a new access method plus all the concurrency, locking, etc. (R-Link-Tree, Bitmaps, External Memory Interval Tree). In contrast to this the enhancing approach is much cheaper, since most properties get inherited, e.g. enhancing B-Trees to be functional B-Trees. We identify the following properties of the integrating approach:



a) Standard ORDBMS kernel

b) Integrating approach

c) Generic approach

d) Relational approach

Figure 6: Approaches to implement custom access methods

Implementation: The implementation of a new AM becomes very sophisticated and tedious if writing transactions have to be supported [Bro01]. 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.

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 too specialized access methods.

Generic Approach. To overcome the restrictions of the integrating method, Hellerstein, Naughton and Pfeffer [HNP95] 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). Many extensions to the GiST framework have been presented, including generic support for concurrency and recovery [KMH97], and additional interfaces for nearest-neighbor search, ranking, aggregation, and selectivity estimation [Aok98]. In detail, the GiST approach has the following characteristics:

Implementation: Whereas the implementation of block-based 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 a high performance for GiST-based access methods. Kornacker [Kor99] 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 only available as a research prototype. It is an open question, if and when a comparable functionality will be a standard component of major commercial ORDBMSs.

The GiST concept basically delivers the desired properties to implement custom access methods. It delegates crucial parts of the implementation to the database vendors. To our best knowledge, however, its full functionality is not available on any major database system. 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 ORDBMS.

Relational Approach. A natural way to avoid the above obstacles is to map the custom 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 [SQL99] 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. In [KPS00] [KPS01] [KMPS01a] [KMPS01b] the capability and efficiency of the relational approach was proven for interval data and 2D/3D spatial data.

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 new 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, custom 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.

3 Basics of Relational Access Methods

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 [SQL92] with common object-relational enhancements in the sense of SQL:1999 [SQL99], including object types and collections.

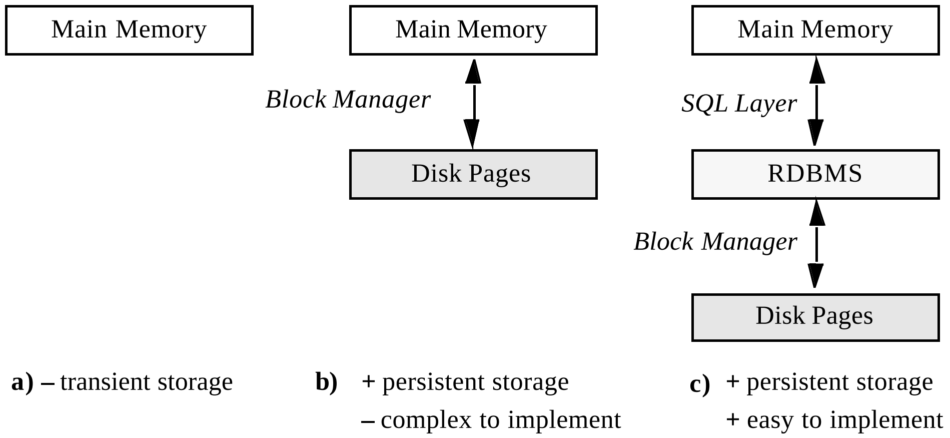
3.1 Paradigms of Access Methods

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 [KP92] [Ora00]. 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 7a). Typical applications of these techniques can be found in main memory databases [DKO+85] [GS92] and in the field of computational geometry [PS93]. A popular example taken from the latter is the binary *Interval Tree* [Ede80]. 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 7b). 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* [AV96] 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 [HNP95], 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 [RMF+00].

Relational Access Methods (Figure 7c). In contrast, relational access methods including the *Relational Interval Tree* [KPS00] are designed to operate on relations rather than on



a) – transient storage

b) + persistent storage
– complex to implement

c) + persistent storage
+ easy to implement

Figure 7: Paradigms and characteristics of access methods: **a)** main memory access methods, **b)** block-oriented access methods, and **c)** relational access methods.

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 [SDF+ 00].

3.2 Relational Storage of Index Data

In the remainder of this paper, 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.

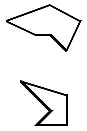
(*iii*) *Meta table*: a single table for each database and each relational access method, storing $O(1)$ rows for each instance of an index.

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

To illustrate the concept of relational access methods, Figure 8 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 8a), com-

polygons

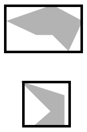
id	geom
A	POLYGON((10,10), (25,15), …, (10,10))
B	POLYGON((30,15), (30,45), …, (30,15))
…	…



a) User table

polygons_mbr

id	mbr
A	BOX((5,10), (30,15))
B	BOX((30,5), (40,50))
…	…



b) Index table

mbr_index_metadata

index_name	user_table	index_table
'polygons_idx'	'polygons'	'polygons_mbr'
…	…	…

c) Meta table

Figure 8: The *MBR-List*, a simple example for a relational access method

prising 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 8b), 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 8c).

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 index data [DDSS95]. This basic approach of relational indexing has already been applied in many existing solutions, including *Linear Quadtrees* [TH81] [RS99] [FFS00] and *Relational R-trees* [RRSB 99] for spatial databases, *Relational X-trees* [BBKM99] for high-dimensional nearest-neighbor search, or inverted indexes for information retrieval on text documents [DDSS95].

4 Operations on Relational Access Methods

In the strict sense of the Definition1, 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[1]. 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

[1] E.g. we replace "blocks.get(*block_id*)" by "select * from blocks where id = *:block_id*".

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 [JS99]. This can be achieved by using declarative operations.

4.1 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 [Sel88] [CD98] [BEKS00]. 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 become very high. For typical applications, the resulting overhead sums up to 30% of the total processing time [RMF+00]. In some experiments, we even reached barrier 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.

4.2 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 semantics 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 [Lib01].

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 [Kor99]. 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 9a 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 9b. 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 9c. 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 [CK97].

```
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 9: Window queries on two-dimensional polygons

5 Generic Schemes for Relational 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.

5.1 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* $\Leftrightarrow$ $(\exists\, t \subseteq T)$ $(\exists\, r_i \subseteq R_i,\ 1 \leq i \leq n)$: at least one $\rho \in r_i$ is associated with rows $\{\tau_1, ..., \tau_m\} \subseteq t$ and $m > 1$.

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* [RRSB99] and the *Relational X-tree* [BBKM99] 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 [SQL99] [EM99] is required.

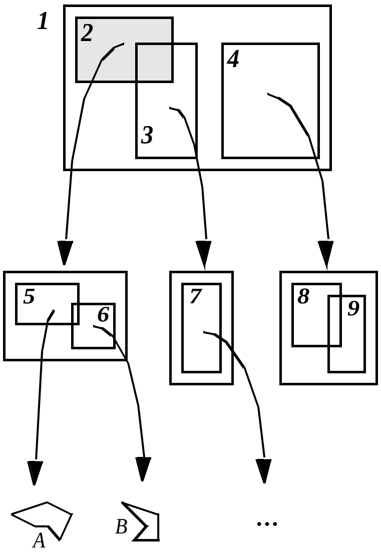
Although the navigational scheme offers a straightforward way to simulate any hierarchical index structure on top of a relational data model, it suffers from the fact that navigational data is locked like user data. As two-phase locking on index tables is too restrictive, the possible level of concurrency 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 [KB95]. 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.

5.2 Relational R-trees – An 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. [RRSB99]. Figure 10 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***,…) is therefore logically associated with the rows (*A*,…) and (*B*,…) in the *polygons* user table (cf. Figure 8). 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 [Ora99c] to support at least concurrent queries on the Relational R-tree index.

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 NF^2 (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 11. We expect that future implementations of the SQL:1999 statement yield a depth-first traversal which is already hard-wired into the existing CONNECTBY clause of the Oracle server. The ef-



polygons_rtree

pageid	page_lev	son_id	son_mbr
ROOT	3	1	BOX((0,0), (200,120))
1	2	2	BOX((0,0), (80,60))
1	2	3	BOX((60,20), (100,120))
1	2	4	BOX((140,20), (200,120))
2	1	5	…
2	1	6	…
5	0	A	…
6	0	B	…
…	…	…	…

a) Hierarchical directory

b) Relational index table

Figure 10: Relational mapping of an R-tree directory

fectiveness 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. If the low concurrency of the Relational R-tree is acceptable, the relational mapping opens up a wide range of potential improvements. We have developed and evaluated various extensions to the presented concept[1]:

- **Variable Fanout.** Due to the relational mapping, we are basically free to allow an individual fanout for each tree node. Similar to the concept of supernodes for high-dimensional indexing [BKK96], larger nodes could be easily allocated, e.g. if the contained geometries show a very high overlap or are almost equal. Thus, splitting such pages would not improve the spatial clustering. Instead, page splits could be triggered by measuring the clustering quality with a proximity measure similar to [KF92]. Especially for CAD databases, where many variants of the same parts occupy almost identical regions of the data space, this approach can be beneficial.
- **Page Clustering.** In order to achieve a good clustering among the entries of each tree node, a built-in primary index can be defined on the *page_id* column. For bulk-loads of Relational R-trees, the clustering can be further improved by carefully choosing the page identifiers: by assigning linearly ordered *page_ids* corresponding to a breadth-first traversal of the tree, a sibling clustering of nodes [KC98] can be very easily achieved.
- **Positive Pruning.** By ordering the *page_ids* according to a depth-first tree traversal, a hierarchical clustering of the R-tree nodes is materialized in the primary index. In consequence, the page identifiers of any subtree form a consecutive range. Similarly, if the

[1] We refer the reader to [Bra00] for detailed descriptions.

```
WITH RECURSIVE tree_traversal (page_lev, son_id, son_mbr) AS (
   SELECT page_lev, son_id, son_mbr FROM polygons_rtree
   WHERE page_id = ROOT
   UNION ALL
   SELECT next.page_lev, next.son_id, next.son_mbr
   FROM tree_traversal prior, polygons_rtree next
   WHERE prior.page_mbr INTERSECTS BOX((0,0),(100,100))
   AND prior.son_id = next.page_id
)                                                      //declarative tree traversal
SELECT son_id AS id
FROM tree_traversal
WHERE page_lev = 0;                                    //select data objects
```

a) Recursive window query on a Relational R-tree using SQL:1999.

```
SELECT son_id AS id FROM polygons_rtree
WHERE page_lev = 0                                     //select data object
START WITH page_id = ROOT
CONNECT BY
   PRIOR son_mbr INTERSECTS BOX((0,0),(100,100))
   AND PRIOR son_id = page_id;                         //declarative tree traversal
```

b) Recursive window query on a Relational R-tree using Oracle SQL.

Figure 11: Cursor-driven window query on a Relational R-tree

leaf pages are hierarchically clustered in a separate B+-tree, a single range query on the *page_id* column yields a blocked output of all data objects stored in any arbitrary subtree of the R-directory. Thus, the recursive tree traversal below a node completely covered by the query region can be replaced by an efficient range scan on the leaf table. Consequently, the tree traversal is not only pruned for all-negative nodes (if no intersection of the node region with the query region is detected), but also for all-positives (the node region is completely covered by the query region). Moreover, heuristics to prune already largely covered nodes can also be very beneficial.

5.3 Direct Scheme of Index Tables

Definition 5 (*Direct 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 *direct* $\Leftrightarrow$ $(\forall\, t \subseteq T)$ $(\forall\, r_i \subseteq R_i, 1 \le i \le n)$: 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 8), the *Linear Quadtree* [Sam90], the one-dimensional *Relational Interval Tree* [KPS00] and its optimization for interval sequences and multidimensional queries [KPS01].

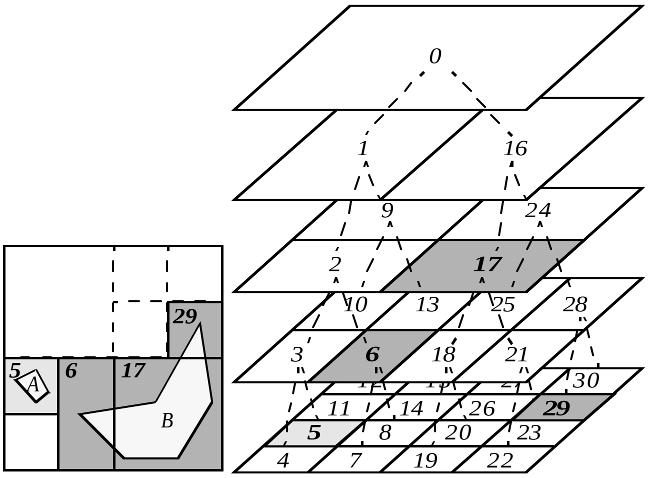
The drawbacks of the navigational scheme with respect to concurrency control and recovery are not shared by the direct scheme, as row-based locking and logging on the index tables can be performed on the granularity of single rows in the user tables. For example, an update of a single row *r* in the user table requires only the synchronization of index rows exclusively 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.

5.4 Linear Quadtrees – An Example for the Direct Scheme

A paradigmatic example for a spatial access method implementing the direct scheme is the *Linear Quadtree* [Sam90]. Several variants of this well-known concept have been proposed for stand-alone balanced trees [TH81] [OM84] [Bay96], for object-oriented database systems [Ore 86] [OM 88] [GR 94], and as relational access methods [Wan91] [IBM98] [Ora99b] [RS99] [FFS00]. In this subsection, we present the basic idea of the Linear Quadtree according to the in-depth discussion of Freytag, Flasza and Stillger [FFS00].

The Linear Quadtree organizes the multidimensional data space by a regular grid. Any spatial object is approximated by a set of *tiles*. Among the many possible one-dimensional embeddings of a grid approximation, the *Z-order* is one of the most popular [Güt94]. The corresponding index representation of a spatial object comprises a set of *Z-tiles* which is computed by recursively bipartitioning the multidimensional grid. By numbering the Z-tiles of the data space according to a depth-first recursion into this partitioning, any set of Z-tiles can be represented by a set of linear values. Note that thereby redundancy is introduced to approximate spatially extended data. Figure 12 depicts some Z-tiles on a two-dimensional grid along with their linear values. The linear values of the Z-tiles of each spatial object can be stored in an index table obeying the schema (*zval*, *id*), where both columns comprise the primary key. This relational mapping implements the direct scheme, as each row in the index table exclusively belongs to a single data object. The linear ordering positions each Z-tile of an object on its own row in the index table. Thus, if a specific row in the user table *polygons* is updated, e.g. (*B*,…), only the rows (***6***, *B*), (***17***, *B*), and (***29***, *B*) in the index table are affected, causing no problems with respect to the native two-phase locking.

In order to process spatial selection on the Linear Quadtree, the query region is also required to be decomposed to a set of Z-tiles. We call the corresponding function *ZDecompose*. For each resulting linear value *zval*, the intersecting tiles have to be extracted from the index table. Due to the Z-order, all intersecting tiles having the same or a smaller size than the tile represented by *zval* occupy the range *ZLowerHull*(*zval*) = [*zval*, *ZHi*(*zval*)] which can be easily computed [FFS00]. In the example of Figure 12, we obtain *ZLowerHull*(17)=[17,23]. In a similar way, we also compute *ZUpperHull*(*zval*), the set of all larger intersecting tiles. As in the case of *ZUpperHull*(17)={0,16} the corresponding linear values usually form no consecutive range. To find all intersecting tiles for



polygons_quadtree

zval	id
5	A
6	B
17	B
29	B
…	…

a) Regular grid **b)** Recursive partitioning **c)** Index table

Figure 12: Relational mapping of a Linear Quadtree

a given *zval*, a range scan on the index table is performed with *ZLowerHull*(*zval*) and multiple exact match queries are executed for *ZUpperHull*(*zval*). These queries are optimally supported by a built-in B+-tree on the *zval* column. Figure 13 depicts the complete cursor-driven window query on an instance of the Linear Quadtree using SQL:1999. Alternatively, the transient rowsets generated by the functions *ZDecompose* and *ZUpperHull* can be precomputed in the procedural phase for all Z-tiles of the query box and passed to the SQL layer in one step by using bind variables. This approach reduces the overhead of barrier crossings between the declarative and procedural environments to a minimum.

6 Conclusions

In this paper, we presented the concept of relational 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

```
SELECT DISTINCT idx.id                                    //select data object
FROM    polygons_quadtree idx,
        TABLE(ZDecompose(BOX((0,0),(100,100)))) tiles,
        TABLE(ZUpperHull(tiles.zval)) uh
WHERE   (idx.zval BETWEEN tiles.zval AND ZHi(tiles.zval))
        OR (idx.zval = uh.zval);
```

Figure 13: Cursor-driven window query on a Linear Quadtree

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 three spatial examples: The *MBR-List*, a trivial relational access method for demonstration purposes, along with the *Relational R-tree* and the *Linear Quadtree*, two fully-fledged spatial access methods implementing the navigational and the direct scheme, respectively.

In our future work we plan to 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.

Acknowledgements. We would like to thank the anonymous referees for their constructive and helpful comments.

References

[Aok 98] Aoki P. M.: *Generalizing "Search" in Generalized Search Trees*. Proc. 14th Int. Conf. on Data Engineering (ICDE): 380-389, 1998.

[AV 96] Arge L., Vitter J. S.: *Optimal Dynamic Interval Management in External Memory.* Proc. 37th Annual Symp. on Foundations of Computer Science: 560-569, 1996.

[Bay 96] Bayer R.: *The Universal B-Tree for multidimensional Indexing*. Technical University of Munich, TUM-I9637, 1996.

[BBKM 99] Berchtold S., Böhm C., Kriegel H.-P., Michel U.: *Implementation of Multidimensional Index Structures for Knowledge Discovery in Relational Databases*. Proc. 1st Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK), LNCS 1676: 261-270, 1999.

[BEKS 00] Braunmüller B., Ester M., Kriegel H.-P., Sander J.: *Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases*. Proc. 16th Int. Conf. on Data Engineering (ICDE): 256-267, 2000.

[BKK 96] Berchtold S., Keim D. A., Kriegel H.-P.: *The X-tree: An Index Structure for High-Dimensional Data*. Proc. 22nd Int. Conf. on Very Large Databases (VLDB): 28-39, 1996.

[Bra 00] Braun C.: *Development and Evaluation of R-Trees for Object-Relational Database Systems* (in german). Diploma Thesis, University of Munich, 2000.

[Bro 01] Brown P.: *Object-Relational Database Development – A Plumber's Guide*. Informix Press, Menlo Park, CA, 2001.

[BSSJ 99] Bliujute R., Saltenis S., Slivinskas G., Jensen C.S.: *Developing a DataBlade for a New Index*. Proc. 15th Int. Conf. on Data Engineering (ICDE): 314-323, 1999.

[CCF+ 99] Chen W., Chow J.-H., Fuh Y.-C., Grandbois J., Jou M., Mattos N., Tran B., Wang Y.: *High Level Indexing of User-Defined Types*. Proc. 25th Int. Conf. on Very Large Databases (VLDB): 554-564, 1999.

[CD 98] Chen F.-C. F., Dunham M. H.: *Common Subexpression Processing in Multiple-Query Processing*. IEEE Trans. on Knowledge and Data Engineering, 10(3): 493-499, 1998.

[CK 97] Michael J. Carey, Donald Kossmann: *On Saying "Enough Already!" in SQL*. Proc. ACM SIGMOD Int. Conf. on Management of Data: 219-230, 1997.

[DDSS 95] DeFazio S., Daoud A., Smith L. A., Srinivasan J.: *Integrating IR and RDBMS Using Cooperative Indexing*. Proc. 18th ACM SIGIR Conference on Research and Development in Information Retrieval: 84-92, 1995.

[DKO+ 85] DeWitt D. J., Katz R. H., Olken F., Shapiro L. D., Stonebraker M., Wood D. A.: *Implementation Techniques for Main Memory Database Systems*. Proc. ACM SIGMOD Int. Conf. on Management of Data: 1-8, 1984.

[Ede 80] Edelsbrunner H.: *Dynamic Rectangle Intersection Searching*. Institute for Information Processing Report 47, Technical University of Graz, Austria, 1980.

[EM 99] Eisenberg A., Melton J.: *SQL:1999, formerly known as SQL3*. ACM SIGMOD Record, 28(1): 131-138, 1999.

[FFS 00] Freytag J.-C., Flasza M., Stillger M.: *Implementing Geospatial Operations in an Object-Relational Database System*. Proc. 12th Int. Conf. on Scientific and Statistical Database Management (SSDBM): 209-219, 2000.

[GHJV 95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns*. Addison Wesley Longman, Boston, MA, 1995.

[GR 94] Gaede V., Riekert W.-F.: *Spatial Access Methods and Query Processing in the Object-Oriented GIS GODOT*. Proc. AGDM Workshop, Geodetic Commission, 1994.

[GS 92] Garcia-Molina H., Salem K.: *Main Memory Database Systems: An Overview*. IEEE Trans. on Knowledge and Data Engineering 4(6): 509-516, 1992.

[Güt94] Güting R. H.: *An Introduction to Spatial Database Systems*. VLDB Journal , 3(4): 357-399, 1994.

[HNP 95] Hellerstein J. M., Naughton J. F., Pfeffer A.: *Generalized Search Trees for Database Systems*. Proc. 21st Int. Conf. on Very Large Databases: 562-573, 1995.

[IBM 98] IBM Corp.: *IBM DB2 Spatial Extender Administration Guide and Reference, Version 2.1.1*. Armonk, NY, 1998.

[IBM 99] IBM Corp.: *IBM DB2 Universal Database Application Development Guide, Version 6*. Armonk, NY, 1999.

[Inf 98] Informix Software, Inc.: *DataBlade Developers Kit User's Guide, Version 3.4*. Menlo Park, CA, 1998.

[Inf 99] Informix Software, Inc.: *Informix R-Tree Index User's Guide, Version9.2*. Menlo Park, CA, 1999.

[JS 99] Jensen C. S., Snodgrass R. T.: *Temporal Data Management*. IEEE Trans. on Knowledge and Data Engineering 11(1): 36-44, 1999.

[KB 95] Kornacker M., Banks D.: *High-Concurrency Locking in R-Trees*. Proc. 21st Int. Conf. on Very Large Databases (VLDB): 134-145, 1995.

[KC 98] Kim K., Cha S. K.: *Sibling Clustering of Tree-based Spatial Indexes for Efficient Spatial Query Processing*. Proc. ACM CIKM Int. Conf. on Information and Knowledge Management: 398-405, 1998.

[KF 92] Kamel I., Faloutsos C.: *Parallel R-trees*. Proc. ACM SIGMOD Int. Conf. on Management of Data: 195-204, 1992.

[KMH 97] Kornacker M., Mohan C., Hellerstein J. M.: *Concurrency Control in Generalized Search Trees*. Proc. ACM SIGMOD Int. Conf. on Management of Data: 62-72, 1997.

[KMPS01a] Kriegel H.-P., Müller A., Pötke M., Seidl T.: *DIVE: Database Integration for Virtual Engineering* (Demo). Demo Proc. 17th Int. Conf. on Data Engineering (ICDE): 15-16, 2001.

[KMPS01b] Kriegel H.-P., Müller A., Pötke M., Seidl T.: *Spatial Data Management for Computer Aided Design* (Demo). Proc. ACM SIGMOD Int. Conf. on Management of Data, 2001.

[Kor 99] Kornacker M.: *High-Performance Extensible Indexing*. Proc. 25th Int. Conf. on Very Large Databases (VLDB): 699-708, 1999.

[KP 92] Keim D. A., Prawirohardjo E. S.: *Datenbankmaschinen – Performanz durch Parallelität*. Reihe Informatik 86, BI Wissenschaftsverlag, Mannheim, 1992.

[KPS 00] Kriegel H.-P., Pötke M., Seidl T.: *Managing Intervals Efficiently in Object-Relational Databases*. Proc. 26th Int. Conf. on Very Large Databases (VLDB): 407-418, 2000.

[KPS 01] Kriegel H.-P., Pötke M., Seidl T.: *Interval Sequences: An Object-Relational Approach to Manage Spatial Data*. Proc. 7th Int. Symposium on Spatial and Temporal Databases (SSTD), LNCS 2121: 481-501, 2001.

[Lib 01] Libkin L.: *Expressive Power of SQL*. Proc. 8th Int. Conf. on Database Theory (ICDT): 1-21, 2001.

[OM 84] Orenstein J. A., Merrett T. H.: *A Class of Data Structures for Associative Searching*. Proc. 3rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS): 181-190, 1984.

[OM 88] Orenstein J. A., Manola F. A.: *PROBE Spatial Data Modeling and Query Processing in an Image Database Application*. IEEE Transactions on Software Engineering, 14(5): 611-629, 1988.

[Ora 99a] Oracle Corp.: *Oracle8i Data Cartridge Developer's Guide, Release 2 (8.1.6)*. Redwood Shores, CA, 1999.

[Ora 99b] Oracle Corp.: *Oracle Spatial User's Guide and Reference, Release 8.1.6*. Redwood Shores, CA, 1999.

[Ora 99c] Oracle Corp.: *Oracle8i Concepts, Release 8.1.6*. Redwood Shores, CA, 1999.

[Ora 00] Oracle Corp.: *Oracle8i Appliance – An Oracle White Paper*. Redwood Shores, CA, 2000.

[Ore 86] Orenstein J.A.: *Spatial Query Processing in an Object-Oriented Database System*. Proc. ACM SIGMOD Int. Conf. on Management of Data: 326-336, 1986.

[PS 93] Preparata F. P., Shamos M. I.: *Computational Geometry: An Introduction*. 5th ed., Springer, 1993.

[RMF+ 00] Ramsak F., Markl V., Fenk R., Zirkel M., Elhardt K., Bayer R.: *Integrating the UB-Tree into a Database System Kernel*. Proc. 26th Int. Conf. on Very Large Databases (VLDB): 263-272, 2000.

[RRSB 99] Ravi Kanth K. V., Ravada S., Sharma J., Banerjee J.: *Indexing Medium-dimensionality Data in Oracle*. Proc. ACM SIGMOD Int. Conf. on Management of Data: 521-522, 1999.

[RS 99] Ravada S., Sharma J.: *Oracle8i Spatial: Experiences with Extensible Databases*. Proc. 6th Int. Symp. on Large Spatial Databases (SSD), LNCS 1651: 355-359, 1999.

[Sam 90] Samet H.: *Applications of Spatial Data Structures*. Addison Wesley Longman, Boston, MA, 1990.

[Sel 88] Sellis T. K.: *Multiple-Query Optimization*. ACM Transactions on Database Systems (TODS), 13(1): 23-52, 1988.

[SDF+ 00] Jagannathan Srinivasan, Souripriya Das, Chuck Freiwald, Eugene Inseok Chong, Mahesh Jagannath, Aravind Yalamanchi, Ramkumar Krishnan, Anh-Tuan Tran, Samuel DeFazio, Jayanta Banerjee: *Oracle8i Index-Organized Table and Its Application to New Domains*. Proc. 26th Int. Conf. on Very Large Databases (VLDB): 285-296, 2000.

[SMS+ 00] Srinivasan J., Murthy R., Sundara S., Agarwal N., DeFazio S.: *Extensible Indexing: A Framework for Integrating Domain-Specific Indexing Schemes into Oracle8i*. Proc. 16th Int. Conf. on Data Engineering (ICDE): 91-100, 2000.

[SQL 92] American National Standards Institute: *ANSI X3.135-1992/ISO 9075-1992 (SQL-92)*. New York, NY, 1992.

[SQL 99] American National Standards Institute: *ANSI/ISO/IEC 9075-1999 (SQL:1999, Parts 1-5)*. New York, NY, 1999.

[Sto 86] Stonebraker M.: *Inclusion of New Types in Relational Data Base Systems*. Proc. 2nd Int. Conf. on Data Engineering (ICDE): 262-269, 1986.

[TH 81] Tropf H., Herzog H.: *Multidimensional Range Search in Dynamically Balanced Trees*. Angewandte Informatik, 81(2): 71-77, 1981.

[Wan 91] Wang F.: *Relational-Linear Quadtree Approach for Two-Dimensional Spatial Representation and Manipulation*. IEEE Trans. on Knowledge and Data Engineering (TKDE) 3(1): 118-122, 1991.