1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

An Extensible Architecture for Supporting Spatial Data in RDBMS

Yun Wang Gene Fuh*
Jyh-Herng Chow Jean Grandbois† Nelson Mattos Brian Tran

Database Technology Institute
IBM Santa Teresa Laboratory
555 Bailey Avenue, San Jose, CA 95141

Abstract

Business intelligence has been the driving force for tight integration of traditional and non-traditional data, such as geographical information, in databases and database applications. Database vendors are finding ways to extend their RDBMS to support these new types of data. In this paper, we present the technology behind the DB2 Spatial Extender that directly supports spatial data types used by geographical information systems (GIS) applications. We also describe our index extensions to the B-tree indexes, so that spatial indexes can be built upon existing B-tree indexes, without changing the underlying B-tree index structures.

1 Introduction

Recently there has been a tremendous interest in business intelligence (BI) to uncover new business information, derive new business rules, and assist decision making, as companies are looking for ways to gain competitive advantages and grow their business. BI has been a driving force for data integration of traditional business data, and non-traditional data, such as geographical information. For example, to select a location for a new store, the decision may depend on many factors that involve geographical data, such as the future growth of the surrounding neighborhood, the distance to highways, and proximity of major competitors.

Most business data today is stored in relational databases. However, traditional RDBMSs can not easily support the new geographical data types required by these new applications. Thus, *geographical information systems* (GIS) have been built as database applications that model, manipulate, query, and analyze spatial data, while handshake with RDBMSs where other business data resides. These GISs rely on a separate data engine, with proprietary APIs, that understands spatial data and is responsible for the management of spatial data and its computations. Figure 1(a) illustrates such GISs. As more

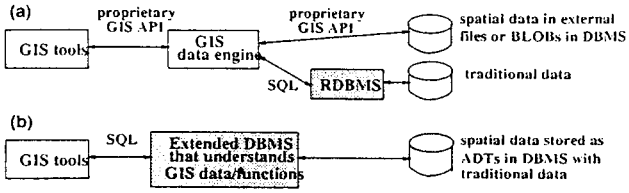

Figure 1: *(a) Traditional GIS systems (b) GIS systems with extended RDBMSs supporting spatial data*

business decisions rely on spatial data, however, these GIS engines become very complicated, ineffective, hard to manage, and unable to react fast to the ever-changing market needs.

At the same time, because of the strong need of storing non-traditional data, such as audio, video and spatial data into databases, several RDBMS vendors are now adding object extensions to their databases. These new extensions allow applications to store and manipulate application objects directly inside the database, and relieve them from dealing with complicated tasks, such as data recovery, backup, and separate query engines. It is now possible to build a tightly integrated consistent system that can exploit the full power of business intelligence through SQL interface. Figure 1(b) illustrates such new systems.

In this paper, we present our work that has been implemented in IBM's DB2 [IBM97] Spatial Extender[1] [DJS98, Dav98] for building such an extended RDBMS that supports spatial data. It relies on two important extensions to traditional RDBMS: *abstract data types* (ADTs) and *user-defined index extensions*. With ADTs, which are now in the forthcoming SQL3 standard [Mel97], user-defined complex objects can be stored in the database, while maintaining individual properties of their fields (instead of black box objects like BLOBs), and user-defined functions (UDFs) can be used to manipulate these objects by the database. In order to efficiently access and search spatial data, which is now stored in the database as ADT objects, we have introduced new extensions to the existing B-tree indexes supported by most

*Contact person: Gene Fuh, IBM Santa Teresa Lab, 555 Bailey Avenue, San Jose, CA 95141. Email: fuh@us.ibm.com. Tel: (408) 463-2661.

†Environmental System Research Institute, 380 New York Street, RedLands, CA 92373

[1] DB2 Spatial Extender is a product jointly developed by IBM and Environmental System Research Institute (ESRI), which is a leading GIS vendor in the world.

RDBMSs. These index extensions provide an extensible index interface that allows various user-defined functions to be integrated into the B-tree index manager for storing and searching, without modifying the underlying B-tree index structures.

There have been other approaches for implementing integrated GIS systems. For example, Informix's DataBlade [Inf97a] allows application-defined index managers to be integrated into their database system. Thus, GIS applications can implement *R-tree* [Gut84, BKSS90], R^+-*tree* [FSR87], *kD-trees* [BF79], or *grid files* [Nie84] as the spatial indexes to be utilized by the database engine. Informix's own Geodetic DataBlade [Inf97b] supports R-tree indexes. Our approach is different from others in the following aspects. First, we do not implement a new index structure and its index manager, which can be very complicated, and requires a deep understanding of underlying database systems to work correctly. Our approach creates spatial indexes directly upon existing B-tree indexes, while opens up several places in the index manager to allow application-specific functionalities. Second, our index extensions are very extensible: updating and searching the index are all user-definable through UDFs. Thus, according to application needs, the user can create various indexes that suit each individual need, without additional supports from the database system.

This paper is organized as follows. In section 2, we will briefly introduce geographical data types, functions, and indexes. In section 3, we describe the existing GIS systems and their weakness. In section 4, we present our implementation of DB2 Spatial Extender for a fully integrated GIS architecture. We describe our spatial data types, and our extensions to the existing B-tree index that allow user-defined functions to be invoked by the index manager. We give the external language specifications of these extensions, without getting into much technical detail, which most likely will be described in a separate paper. Section 5 gives an example of typical GIS applications, as a thorough work-through about how the extensions are used in the DB2 Spatial Extender. Section 6 concludes the paper and discusses some future work.

2 Spatial Data, Functions, and Indexes

Geographical information is described by a coordinate geometry and a reference system. The coordinate geometry consists of the number of dimensions, the coordinates, and a sequence of coordinate points in the same reference system. Data from different data sources likely will have different reference systems; thus, their integration and manipulation require translations to a common reference system. Geometry data is built using points (0-dimension), lines (1-dimension), polygons (2-dimension), polyhedra (3-dimension), and even spatial-temporal data (4-dimension) Readers may want to consult the documents by *Open GIS Consortium* [Ope98a] for a complete specification. In this work, we only focus on geometry of dimensions 0, 1, and 2, and we will use *spatial* data to refer to such geometry data.

Real world geographical data is represented using these basic spatial data types. For example, a park site is represented by the location of its geographical center, a river system is represented by a set of sample points along the path, and a county is represented by a polygon that approximately matches the boundary of the county. With the basic spatial attributes being collected and modeled, complex spatial information such as the length or area of a geographical object can be derived. Moreover, spatial relationships between objects can be computed, for example, *intersection*, *within*, *distance*, *contain*, *touch*, and so on [Ope98b]. These derived information, together with data mining, often provides tremendous values to decision support systems.

The *envelope* of a geometry is the bounding geometry formed by a maximum and minimum coordinates. It is a rectangle, called *minimal bounded rectangle* (MBR), except the case of points where the envelope of a point is the point itself. An envelope is a simple approximation of a geometry, and is good for fast approximate evaluations as early filters to reduce the cost in full evaluations. For example, if the envelopes of two objects do not overlap (easy to compute by comparing the coordinates), then certainly the objects do not overlap (hard to compute).

Indexes are necessary for fast searches of spatial data. A set of access methods and evaluation functions that take advantages of the spatial index must also be implemented. R-trees, grid files, and their variations are commonly used structures for spatial indexes. R-trees [Gut84, BKSS90] and R^+-trees [FSR87] are based on B-trees, where the leaf nodes contain actual rectangular data and a non-leaf node corresponds to an MBR that contains the rectangles of its children. Overlaps of rectangles described by non-leaf nodes are also allowed. On the other hand, indexes based on grid files divide the space into grids of either fixed-sized cells [BF79], or unequal-sized cells [Nie84]. The grid cells that intersect with the given object (or the object's envelope) become the index entries of the object. Multiple layers of index grids can also be used to provide different resolutions, so that fast evaluations can be done at a low resolution layer.

The index extensions proposed in this work do not assume a specific type of spatial index structures. The actual implementation in the DB2 Spatial Extender, however, uses multiple layers of fix-sized grid indexes *logically* (physically they are B-tree indexes). An index key is in fact denoted by the following tuple `<level,gx,gy,xmin,ymin,xmax,ymax>`, where `level` is the grid layer, `gx` and `gy` are the x- and y-coordinates of the lower-left corner of a grid cell, and the last four values are the coordinates of the lower-left corner and the upper-right corner of the MBR of the object overlapped with the grid cell. This kind of index keys has the following characteristics. First, more than one index entries, denoting different layers and grid cells, can be associated with a geometry object. Second, index keys with the same grid cell `<level,gx,gy>` denote objects that overlap with the cell. Thus, the index can be used to evaluate range queries asking for objects that are located within a certain range. Third, the index keys are not unique, because

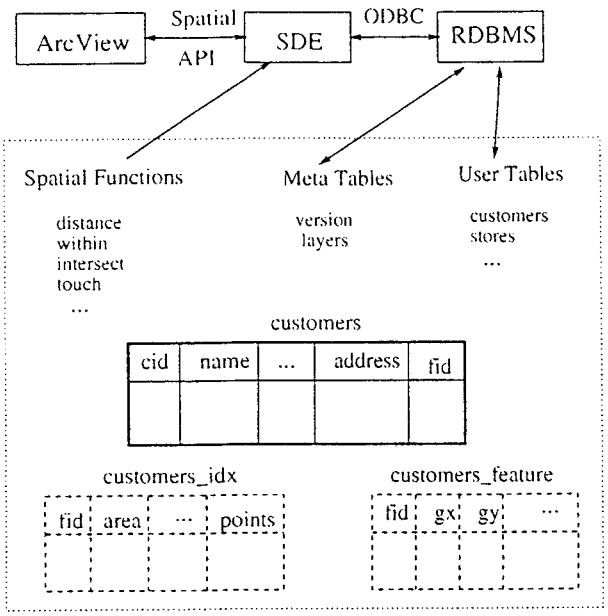

Figure 2: *A typical existing GIS architecture*

more than one object may have the same MBR.

We will describe the creation and usages of this index in section 5.

3 Existing GIS Systems

Figure 2 shows a typical GIS architecture in the market today. We use ESRI's ArcView [ESR] as our example. A set of proprietory spatial functions and predicates are maintained by the Spatial Data Engine (SDE). The RDBMS must be first enabled by external commands for spatial support, which will create meta tables for the administration of spatial operations. Each spatial column in a business table is represented by a unique feature ID (indicated by fid) and is associated with a couple of side tables: the feature table that contains the spatial information for the spatial column, and the index table that contains the index entries of the spatial data.

A spatial query is composed by the user through ArcView GUI interface, and sent to SDE using non-standard spatial APIs. A spatial query contains spatial constraints in addition to normal database queries, as shown in Figure 3.

SDE processes the query by transforming the spatial constraints into a complicated set of SQL predicates, involving the side tables and spatial predicates, to exploit the spatial index, as shown in the figure. The resulting complicated SQL query is then sent to RDBMS via ODBC interface, and finally the RDBMS compiles and executes the query without knowing that the query is a spatial query.

Due to the lack of tight integration with RDBMS, this architecture results in the following weakness:

- Data integrity: The spatial information is still not directly integrated into other business data. Spatial

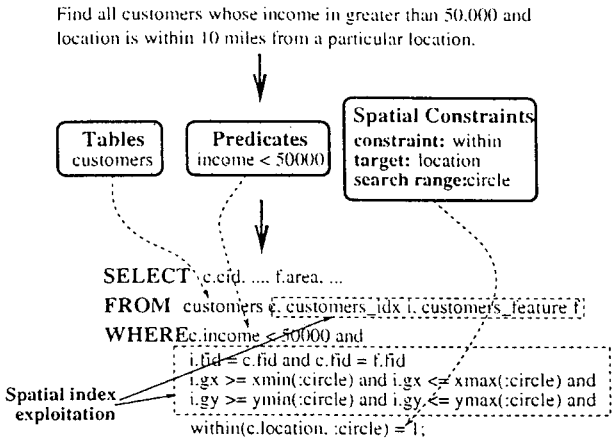

Figure 3: *A spatial query and its transformation*

data are manipulated outside of RDBMS, and thus various data integrity problems, such as error recovery and rollback, can not be easily maintained.

- Ineffective use of storage: Because computation involving spatial data are carried out in the GIS engine, data must be moved out of the RDBMS, and in fact data is often duplicated in the GIS engine.
- Usability and complexity of maintenance: Two side tables are created for every spatial column, and their integrity must be maintained in sync with the business table, for example, by triggers.
- Complexity of the GIS engine: All the computation and management of spatial data are done by the GIS engine. The GIS engine manages the spatial indexes by itself, and needs a smart query optimizer to exploit spatial indexes by transforming a spatial query into a SQL query, which hopefully is compiled by the RDBMS as expected, although there is no such a guarantee that the RDBMS will choose a plan anticipated by the GIS engine.
- Performance: The RDBMS is unaware of the spatial data it is dealing with, and thus may choose a sub-optimal join order. Furthermore, the GIS engine maintains its own indexes and can not exploit the performance benefits that integration with the RDBMS can offer, such as predicate evaluations, and results in huge amounts of data movement.

4 Extensions to RDBMSs for Supporting Spatial Data

The lack of tight integration with the RDBMS data has been a major obstacle for GIS vendors to provide their services to large IT organizations. The following are some key issues that must be addressed before we can design or enhance a RDBMS that understands spatial data:

1. storing spatial data: how to store the non-relational data into relational databases, and still maintain the spatial properties in a reasonable and effective way?

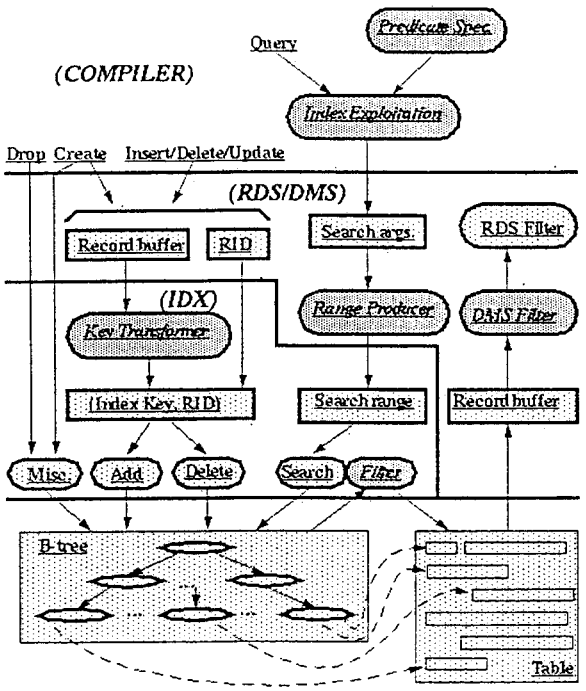

Figure 4: *The gray boxes, Key Transformer, Range Producer, Filter, and DMS Filter, are the places that a UDF can be plugged in through the* `CREATE INDEX EXTENSION` *statement.*

2. invoking spatial functions and predicates: how does the RDBMS invoke spatial functions and predicates in addition to its relational functions? and can they be executed in an optimal way?
3. index structures: what kind of index structures that must be supported in order to achieve good performance on spatial data?
4. index exploitation: how to teach the RDBMS compiler to exploit spatial indexes? can it achieve global optimization of spatial and non-spatial data accesses?

In this section, we will present our architecture (shown in Figure 4) that fully integrates spatial data into relational databases and solves the weakness in current GIS systems. This architecture has been implemented in the DB2 Spatial Extender [DJS98] that directly integrates GIS functionality into the database engine. Our architecture relies on two important extensions to the traditional RDBMS:

- Abstract data type: this allows us to store complex data in an organized way that co-exists with other relational data, and most importantly, it can be managed by the RDBMS consistently and effectively.
- Index extension: this allows user-defined indexes to be created inside the database, using the traditional B-tree indexes that most RDBMSs support.

We will present this architecture in the context of DB2, although we believe a similar design of the architecture can be implemented in other RDBMSs.

4.1 Spatial Data Types

Abstract data types are new features in the SQL3 standard, and have been supported by several relational databases with object extension. With ADTs, complex objects in applications can be stored in and manipulated by the database engine directly, while the object hierarchy and inheritance being observed, for example, by UDF invocations and index accesses.

. ADTs are defined by the `CREATE ADT` statements and subtypes are defined by the `UNDER` clause as shown below.

```
CREAT ADT shape (gtype char(1),area float,length float,
        xmin float, ymin float, xmax float, ymax float,
        numOfParts int, numOfPoints int,
        geometry blob(1M));
CREATE ADT point UNDER shape;
CREATE ADT line UNDER shape;
CREATE ADT polygon UNDER shape;
CREATE TABLE customers (cid int, name varchar(20),
          address varchar(50), income float,   ...
          location point LENGTH 100,
          zone polygon LENGTH 1000);
```

Here we create the `shape` ADT, and its subtypes, `point`, `line`, and `polygon` for our spatial data, and how they are used in the `customers` table. The `gtype` attribute indicates whether the associated entity is a point, a line or a polygon. The `xmin`, `ymin`, `xmax`, and `ymax` attributes model the minimum bounded rectangle of the spatial entity. The actual geographical boundary of the spatial entity is modeled by attributes `numOfParts`, `numOfPoints`, and `geometry`.

ADTs allow business data and spatial data to be stored together naturally in a table, and remove the need for the feature table presented earlier. Moreover, the type hierarchy of spatial data represented in the database reflects the semantics expected by the application.

4.2 Index Extensions

Once we have stored the spatial data into the database, the next question is how we can access them efficiently. In DBMSs, indexes are often created for fast retrieval of data that is frequently used. An index support normally involves three main tasks: index maintenance (e.g., create, drop, insert, update, and delete operations), index search (e.g., range queries), and index exploitation (e.g., determining predicate evaluation order, with the interaction from other database operations).

Traditional index structures supported by RDBMS are B-tree or its variation, and in the case of DB2, B-tree is the only index structure available. Although B-tree indexes have been extensively tested and enhanced, they are only able to deal with basic data types, such as integers or character strings. Naively coercing complex objects into these B-tree indexes is often awkward and results in bad performance.

<create index extension> ::= CREATE INDEX EXTENSION <header> <index maintenance> <index search>
<header> ::= <indexExtensionName> ({ <parmName> <parmType> }*)
<index maintenance> ::= WITH INDEX KEYS FOR ({ <colName> <colType> }*)
GENERATED BY <function invocation>
<index search> ::= WITH SEARCH METHODS FOR INDEX KEYS ({ <colName> <colType> }*) <method>
<method> ::= WHEN <methodName> USING ({ <colName> <colType> }*)
RANGE THROUGH <function invocation> CHECK WITH <function invocation>
<drop index extension> ::= DROP INDEX EXTENSION <indexExtensionName>
<create index> ::= CREATE [UNIQUE] INDEX <indexName> ON <tableName> ({ <colName> [ASC | DESC] }*)
USING <indexExtensionName> ({ <constant> }*)

Figure 5: *Language specifications for index extensions. N^* specifies one or more occurrence of N, with separator ','.*

In order to access spatial data and evaluate spatial predicates efficiently, we have introduced general user-defined extensions to the B-tree functionality, without changing the underlying B-tree index structure. This results in two major advantages of this approach. First, we can use the existing index structures to support new kinds of data types, without requiring a completely new index manager, which can be very costly to implement in a matured commercial product. Second, this general approach allows various index functions to be user-definable. This can offer significant performance improvement when domain-specific predicates can be pushed deep down into the index manager.

We introduced the concept of index extension specification that supports parametric user-defined index types, using the `CREATE INDEX EXTENSION` statement. Figure 5 shows the language specifications for these index extensions. This statement defines an index extension over the existing B-tree indexes, and acts like a template that can be instantiated at index creation time by the CREATE INDEX statement into different index instances. Essentially, this statement associates user-defined behaviors to various functions of an index manager that we have opened up and parameterized. Figure 4 shows these extensions in the index manager, indicated by gray boxes.

The *<header>* specifies the index extension name and a list of instance parameters, which will be used in the new USING clause of a CREATE INDEX statement. For example, we can create an index extension for spatial data, and its instance parameters specify multiple grid levels and allow different grid sizes to be used in an index that uses this index extension.

For index maintenance, since now an index can be defined over a complex object type, to use the B-tree index, the user can define how B-tree index entries are computed from the complex objects through UDFs. This is specified by the GENERATED BY clause, and is indicated by the key transformer box in the Figure 4. For index search, we allow a user-defined range producer (specified by RANGE THROUGH clause) that specifies how a search argument is translated to search key ranges of a B-tree index. This is indicated by the range producer box. Furthermore, in addition to the normal B-tree search methods, we allow further user-defined search methods (through CHECK WITH clause) to be invoked by the index manager during an index search, in order to filter out non-qualified tuples as early as possible. These are indicated by the Filter box. For index exploitation, we modify the DB2 optimizer to understand spatial predicates, and to generate an access plan that globally optimizes both spatial and non-spatial data accesses. And finally, to exploit indexes during predicate evaluations, a list of search methods can be specified in the WITH SEARCH METHODS clause, indicated by the predicate spec box. These search methods, coupled with the new extensions added to the `CREATE FUNCTION` statement, allow fine-tuned user-defined functions to be invoked for further data filtering, indicated by the DMS filter box, depending on the context where the function is used as a predicate.

After an index extension has been created, the user then can create an index using the index extension through the new USING clause in the CREATE INDEX statement, and afterwards, all user-defined functions will be invoked at appropriate places along an index operation.

Figure 6 shows the extension to user-defined functions. The CREATE FUNCTION statement now can have a list of predicate specifications. Each predicate specification defines when (the matching context) the UDF being defined is considered as a predicate and, if so, how (data filter) can the optimizer optimize the execution of this UDF and how (index exploitation) this UDF can be used to exploit indexes. The matching context is specified by the AS PREDICATE WHEN clause, the data filter is specified by the FILTER BY clause, and the index exploitation rule is specified by the WHEN KEY *<paramName>* USE *<methodName>*(...) clause, where *<methodName>* is a method defined in the index extension *<indexExtensionName>*.

The following example defines a UDF, `within`, and it is used in a SELECT statement. The `within` function is defined as a predicate if its result is compared with the constant 1; thus, the optimizer will consider this predicate to exploit indexes. Furthermore, since the evaluation of the second argument (`circle(:x, :y, :r)`) in the query can be performed before the tuple is fetched from the customer table, it can be picked up by the optimizer as a search argument. Similarly, the first argument is a simple column from the customer table. Therefore, the optimizer will consider it as a search target.

<*create function*> ::= CREATE FUNCTION <*functionName*> { <*parmName*> <*dataType*> }* <*predicate specification*>*
<*predicate specification*> ::= AS PREDICATE WHEN <*comparisonOp*> <*constant*>
[FILTER BY <*function invocation*>] [<*index exploitation*>]
<*index exploitation*> ::= SEARCH BY INDEX EXTENSION <*indexExtensionName*> <*exploitation rule*>*
<*exploitation rule*> ::= WHEN KEY ({ <*paramName*> }*) USE <*methodName*> ({ <*paramName*> }*)

Figure 6: *Language specifications for user-defined functions*

```
CREATE FUNCTION within(x shape, y shape)
RETURNS INT
LANGUAGE C
EXTERNAL NAME `/u/fuh/db2sde/gis!within'
AS PREDICATE WHEN = 1
FILTER BY mbrOverlap(x..mbr, y..mbr)
SEARCH BY INDEX EXTENSION grid
WHEN KEY(x) USE searchFirstBySecond(y);

SELECT * FROM customer
WHERE within(location, circle(:x,:y,:r)) = 1;
```

To summarize the extensions to the B-tree index manager, there are four places that UDFs can be invoked to apply application-specific operations:

- **Key Transformer**: the UDF specified in the GENERATED BY clause. This is the only one that is applied at insert/update/delete time. The others are applied at query time. Given a record, a UDF for key transformer (a table function[2]) can be invoked to generate a set of keys to be used by the index manager for index maintenance. Note that multiple entries can exist in the index for a single record.
- **Range Producer**: the UDF specified in the RANGE THROUGH clause. Given a user search predicate, a UDF for range producer (a table function) can be invoked to generate a set of start/stop key pairs for searching in the B-tree index.
- **IDX Filter**: the UDF specified by the CHECK WITH clause. This boolean filter UDF is invoked right after a RID is retrieved from the index. A common use of this UDF is to remove duplicates, because multiple entries may exist for a RID.
- **DMS Filter**: the UDF specified by the FILTER BY clause of CREATE FUNCTION statement. This boolean filter UDF is invoked after the RID is used to retrieve the data record and before the original predicate is applied. This adds another filter operation early in the index operations right after non-indexed column values are fetched.

5 A Complete Example

In this section, we will go through a real scenario of modeling/indexing spatial data using the new extensions. We will focus on the creation and exploitation of spatial index, and leave out other issues such as bind-in/bind-out of spatial data and implementation details of user-defined functions.

[2] A user-defined table function is a UDF that returns a set of tuples [IBM97].

5.1 Modeling Spatial Data

The following `create adt` statements define an envelope, and a type hierarchy of geometric shapes, with `shape` as the supertype and `nullshape`, `point`, `line`, and `polygon` as its subtypes. Each instance of these ADTs models a spatial entity such as the location of a store (point), the path of a river (line), or the boundary of a business zone (polygon). The `gtype` attribute indicates whether the associated entity is a point, a line, or a polygon. The `mbr` attribute models the minimum bounded rectangle of the spatial entity. The geographical boundary of the spatial entity is modeled by attributes `numpart`, `numpoint`, and `geometry`. The `refsystem` attribute specifies the reference system.

```
create adt envelope (xmin int, ymin int,
                     xmax int, ymax int);
create adt shape not instantiable (
        gtype     varchar(20),
        refsystem int,
        mbr       envelope,
        numpart   sint,  numpoint  sint,
        geometry  blob(1m));
create adt nullshape under shape;
create adt point     under shape;
create adt line      under shape;
create adt polygon   under shape;
```

5.2 Index Extension `grid_extension`

The statement in Figure 7 creates an index extension, named `grid_extension`, with a instance parameter `levels` for grid levels. This index extension specifies the index is for `shape` data type, whose entries are generated by the UDF `gridEntry` (as the key transformer). It has two search methods, `searchFirstBySecond` and `SearchSecondByFirst`, each of which specifies its range producer and index filter UDF. See Table 1.

5.2.1 Key Transformer `gridEntry`

The key transformer UDF takes as input the `mbr` attributes of the index column (of type `shape`) and the instance parameter (of type `varchar(20) for bit data`) and generates as output a set of index keys, where each key consists of 7-tuple, as described in section 2. Recall that a shape may generate multiple index keys; thus, the result of this UDF is a set of keys.

search method	search target	range producer	IDX filter	description
searchFirstBySecond	shape	gridRange	checkDuplicate	Used for the within predicate where the first argument is the search target and the second argument is the search argument.
searchSecondByFirst	shape	gridRange	mbrOverlap	Used for the within predicate where the second argument is the search target and the first argument is the search argument.

Table 1: Search methods for grid_extension.

```
create index extension grid_extension(levels varchar(20) for bit data)
with index keys for (shapeCol shape)
generated by gridEntry(shapeCol..mbr..xmin, shapeCol..mbr..ymin,
                       shapeCol..mbr..xmax, shapeCol..mbr..ymax, levels)
with search methods for index keys (level int, gx int, gy int, xmin int, ymin int, xmax int, ymax int)
when searchFirstBySecond using (searchArg shape)
     range through gridRange(searchArg..mbr..xmin, searchArg..mbr..ymin,
                             searchArg..mbr..xmax, searchArg..mbr..ymax, levels)
     check with checkDuplicate(level, gx, gy, xmin, ymin,xmax, ymax, searchArg..mbr..xmin,
                               searchArg..mbr..ymin, searchArg..mbr..xmax, searchArg..mbr..ymax, levels)
when searchSecondByFirst using (searchArg shape)
     range through gridRange(searchArg..mbr..xmin, searchArg..mbr..ymin,
                             searchArg..mbr..xmax, searchArg..mbr..ymax, levels)
     check with mbrOverlap(xmin,ymin,xmax,ymax, searchArg..mbr..xmin, searchArg..mbr..ymin,
                                                searchArg..mbr..xmax, searchArg..mbr..ymax);
```

Figure 7: *Create index extension* grid_extension

```
create function gridEntry (xmin int, ymin int,
   xmax int, ymax int, levels varchar(20) for bit data)
returns table (level int, gx int, gy int, xmin int,
               ymin int, xmax int, ymax int)
language c      parameter style db2sql
not variant not fenced no sql no external action
external name `/u/fuh/db2sde/gis!gridEntry´;
```

Most of the clauses in a CREATE FUNCTION statement are not of our interest (see [IBM97] for the detailed description). The external name clause points to the C implementation of this UDF.

5.2.2 Range Producer gridRange

The range producer UDF takes an mbr and the instance parameter, and returns a set of search ranges, where each range is specified by a start and stop key.

```
create function gridRange (xmin int, ymin int,
              xmax int, ymax int, levels varchar(40))
returns table (slevel int,   // Start of the range
               sgx    int, sgy    int,
               sxmin  int, symin  int,
               sxmax  int, symax  int,
               elevel int   // Stop of the range
               egx    int, egy    int,
               exmin  int, eymin  int,
               exmax  int, eymax  int);
language c       parameter style db2sql
not variant not fenced no sql no external action
external name `/u/fuh/db2sde/gis!gridRange´;
```

5.2.3 IDX Filter checkDuplicate and mbrOverlap

The IDX filter UDFs are invoked as filters right after keys in search ranges are retrieved from the index. Since a single shape object may produce multiple keys in the index, we use checkDuplicate and mbrOverlap to filter out keys that are from the same objects.

```
create function checkDuplicate (gx int, gy int,
     x1 int, y1 int, x2 int, y2 int, xmin int,
     ymin int, xmax int, ymax int, level varchar(40))
returns integer
language c        parameter style db2sql
not variant not fenced no sql no external action
external name `/u/fuh/db2sde/gis!checkDuplicate´;

create function mbrOverlap (xmin1 int, ymin1 int,
      xmax1 int, ymax1 int, xmin2 int, ymin2 int,
      xmax2 int, ymax2 int)
returns integer
language c        parameter style db2sql
not variant not fenced no sql no external action
external name `/u/fuh/db2sde/gis!mbrOverlap´;
```

5.3 Creating Spatial Predicate within

We now create the spatial predicate, within, that tests whether one shape is within another.

```
create function within (shape, shape)
returns integer
language c         parameter style db2sql
not variant not fenced no sql no external action
external name `/u/fuh/db2sde/gis!within´
```

```
as predicate when = 1
   filter by mbrWithin(x..mbr..xmin, x..mbr..ymin,
                       x..mrb..xmax, x..mbr..ymax,
                       y..mbr..xmin, y..mbr..ymin,
                       y..mrb..xmax, y..mbr..ymax)
   search by index extension grid_extension
          when key(x) use searchFirstBySecond(y)
          when key(y) use searchSecondByFirst(x);
```

This spatial predicate specifies `mbrWithin` as the DMS filter. It also specifies that a spatial index can be exploited when the result of this predicate is tested against 1. If so, the index extension `grid_extension` and its associated UDFs will be used. In this case, the search method `searchFirstBySecond` is used (i.e., with the `gridRange` and `checkDuplicate` UDF) when `x` is a key, or the search method `searchSecondByFirst` is used (with `gridRange` and `mbrOverlap` UDF) when `y` is a key.

The `mbrWithin` is just a normal UDF to test whether one `mbr` is within another.

```
create function mbrWithin(int, int, int, int,
                          int, int, int, int)
returns integer
language c        parameter style db2sql
not variant not fenced no sql no external action
external name `/u/fuh/db2sde/gis!mbrWithin´;
```

5.4 Creating Application Tables and Spatial Indexes

We now are ready to create the application tables that use our spatial data type, and create spatial indexes for them.

```
create table store(sid   int, name  varchar(10), ...
            loc  point check(loc..refsystem = 1),
            zone  shape check(zone..refsystem = 1));
create index store_zone on store(zone)
using grid_extension(`10 100 1000´);

create table customer(cid int, name varchar(10), ...
            income int, addr   char(20),
            loc  shape check(loc..refsystem = 1));
create index customer_loc on customer(loc)
        using grid_extension(`10 100 1000´);
```

The `store` table has a column `loc` of the `point` type, and a column `zone` of the `shape` type. Both specify the check constraint that its `refsystem` must be 1. An index is created on the `zone` column, using the index extension named `grid_extension`, with levels `´10 100 1000´`. This essentially creates a spatial index on the `zone` column with three grid sizes (10, 100, and 1000).

Similarly, the `customer` table has a `loc` column of shape type, and a spatial index is created over the column with the same index extension.

5.5 Spatial Query, Index Exploitation

Finally, we issue a query that uses our spatial predicates. We want to find out those customers that are located close to the location (100,100), or close to the location (300,300). The UDF `circle` creates a `shape` object, where the center of the circle is specified by the first and second argument, and the radius is by the third argument.

```
select * from customer
where within(loc, circle(100, 100, 10)) = 1 or
      within(loc, circle(300, 300, 20)) = 1;
```

The query compiler takes several steps to generate the access plan with spatial index exploited for the above spatial query:

- The function invocation `within(...) = 1` is indeed a predicate (as defined in the predicate specification part of the create function statement) and, hence, needs to be treated differently.
- Based on the information provided in the predicate specification, the query compiler understands how to exploit the spatial index. For both predicates, the first argument is the search target, as it is a spatial column. The second argument is the search argument, as it is not dependent on the evaluation of the rest of the query. Therefore, the search method `searchFirstBySecond` of the index extension `grid_extension` is picked up as the search method.
- The `searchFirstBySecond` indicates that the user-defined table function `gridRange` is to be used for generating search ranges for the search target. Furthermore, the UDF `checkDuplicate` will be invoked right after the index keys are fetched.
- The query compiler will generate an access plan that index-scans the `customer` table twice, one for each `within` predicate, and results are or'ed together. The object created by each `circle` call is used to test whether a customer location is within that circle.
- For both `within` predicates, the mbr of the search argument is computed and `gridRange` is called to produce the start/stop key ranges for the B-tree index accesses.
- For each index entry within the range, the UDF `checkDuplicate`, is invoked to make sure no duplicate entry is returned; duplicates are possible due to that a given spatial object can overlap with more than one grid blocks.
- Once the index entries are collected by the index-scan operation (actually, index ORing in this example), the record IDs are used to fetch the table records from the `customer` table into the buffer pool. The filter function, `mbrWithin`, is then invoked to further filter out un-qualified tuples.
- Finally, the resulting set of tuples is evaluated against the (residual) predicates in the `where` clause of the query to return the result.

Let's conclude this section by considering a slightly different scenario:

```
select * from customer c, store s
where within(c.loc, s.zone) = 1;
```

In the above example, both arguments of the predicate are columns, but from two different tables. The `within` predicate is obviously a join-predicate. The index exploitation process is very similar to that of the previous example. The only difference is that, to exploit the spatial index defined over the `loc` column of the `customer table`, the optimizer will likely pick up the `store` table as the outer of a nest-loop join plan, so that the inner is just a probe to the spatial index.

6 Conclusion

Geographical information has enabled many applications to assist business decisions. However, existing geographical information systems, for the lack of tight integration of spatial data into relational databases, are getting very complicated, ineffective, hard to manage, and slow to react to the market needs. Today, several RDBMS vendors try to provide tight data integration of spatial data into relational databases by adding new functions for GIS applications. In this paper, we describe the work implemented in DB2 Spatial Extender for such an effort. It relies on two extensions to the traditional RDBMSs: abstract data types and user-defined index extensions to the existing B-tree indexes.

We described the specifications of index extensions that allow user-defined functions to be invoked by the index manager at several stages of an index update and search. We also described new specifications to user-defined functions that help index exploitations. We then gave a complete example to explain how these extensions work together to provide an efficient, integrated database interface for GIS applications. Preliminary performance gains by the integrated approach have achieved up to 3X improvements over the existing non-integrated one.

The presented index extensions are very extensible, and we are currently investigating how other DB2 extenders can be simplified by using such extensions. We would also like to investigate the possibilities of opening up more places, in addition to those presented, in the index manager for plugging in user-defined functions. Another area for future work is to enhance the cost model of the database optimizer, so that more accurate cardinality estimates can be made for spatial predicates.

Acknowledgement

The authors would like thank the members of IBM's DB2 Spatial Extender team, and ESRI's developers who worked with us and helped us in understanding GIS applications.

References

[AG97] N.R. Adam and A. Gangopadhyay. *Database Issues in Geographic Information Systems.* Kluwer Academic Publishers, 1997.

[BF79] J.L. Bentley and J.H. Fridman. Data Structures for Range Searching. *ACM Computing Surveys*, 11(4), 1979.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R-tree: An efficient and robust access method for points and rectangles. In *19 ACM SIGMOD Conf. on the Management of Data, Atlantic City*, May 1990.

[Dav98] Judith R. Davis. IBM's DB2 Spatial Extender: Managing Geo-Spatial Information Within the DBMS. 1998. http://www.software.ibm.com/data/pubs/ papers/#spatial.

[DJS98] IBM DB2 Spatial Extender Administration and Reference. 1998.

[ESR] Environmental System Research Institute (ESRI). Home page http://www.esri.com.

[FSR87] C. Faloutsos, T. K. Sellis, and N. Roussopoulos. Analysis of object-oriented spatial access methods. In *19 ACM SIGMOD Conf. on the Management of Data*, May 1987.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, June 1984.

[IBM97] IBM Corporation. *IBM DB2 Universal Database Version 5, SQL Reference*, 1997. Also http://www.software.ibm.com/cgi-bin/db2www/library/pubs.d2w/ report#UDBPUBS.

[Inf97a] Informix. *Informix DataBlade Products*, 97. http://www.informix.com/informix/products/ answers/oldsite/answers/english/datablad.htm.

[Inf97b] Informix. *Informix Geodetic DataBlade Module User's Guide, Version 2.1*, 97. http://www.informix.com/informix/products/ answers/oldsite/answers/pubs/pdf/3711.pdf.

[Mel97] Jim Melton, editor. *(ANSI/ISO Working Draft) Funcation (SQL/Foundation).* International Organization for Standardization and American National Standards Institute, x3h2-97-315/dbl:bbn-008 edition, 1997.

[Nie84] J. Nievergelt. The Grid File: An Adaptable, Symmetric, Multikey File Structure", for points and rectangles. *ACM Transaction on Database Systems*, 9(1), 1984.

[Ope98a] OpenGIS Consortium Inc. *OpenGIS Abstract Specification and OpenGIS Implementation Specification*, 1998. http://www.opengis.org/techno/specs.htm.

[Ope98b] OpenGIS Consortium Inc. *OpenGIS Simple Features Specification for SQL*, 1998. Revision 1.0. Available at http://www.opengis.org/public/sfr1/ sfsql_rev_1_0.pdf.