High Level Indexing of User-Defined Types

Weidong Chen*, Jyh-Herng Chow, You-Chin (Gene) Fuh, Jean Grandbois[1]
Michelle Jou, Nelson Mattos, Brian Tran, Yun Wang

IBM Santa Teresa Laboratory
[1]Environmental System Research Institute

Abstract

To support emerging database applications, object-relational databases leverage the mature relational database technology and allow users to introduce application-specific types and methods. Tables in a database may now contain such objects as geographical shapes, images, and text documents. To realize the full potential of object-relational databases, efficient querying and searching of user-defined objects must be supported. This paper presents a high level framework for indexing of user-defined types with user-defined predicates. It is orthogonal to the low level access methods that are supported. It is unique in providing direct user control over the transformation from a user-defined type into an abstract domain of index key values, the generation of search keys of a user-defined predicate with bound search arguments, and the execution of a user-defined predicate. The high level framework has been implemented in IBM DB2 Universal Database. Its generality, usability, and performance have been demonstrated in different application domains, where indices on user-defined objects can be fully integrated with SQL queries and exploited by the query compiler.

*Contact author: Weidong Chen, IBM Santa Teresa Laboratory, 555 Bailey Ave, Room C347, San Jose, CA 95141. Phone: (408) 463-2443. Fax: (408) 463-3834. Email: cwd@us.ibm.com. On sabbatical leave from Southern Methodist University.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Emerging database applications require scalable management of large quantities of complex data together with traditional business data and flexible querying capabilities for business intelligence. Such applications are called *universal applications* [25]. Object-relational databases have been developed to support universal applications. They leverage the mature relational database technology, which provides scalability, reliability, and recovery. More importantly object-relational databases enable users to introduce application-specific types and methods into a database. Tables in a database may now contain such user-defined objects as geographical shapes, images and semi-structured text documents.

The business value of complex data cannot be fully realized unless efficient search and querying can be provided on user-defined objects together with the traditional business data. Unfortunately existing commercial databases are rather primitive in their support for access and indexing of user-defined objects. B-trees [3, 9] often serve as the sole indexed access method (although Informix [18] does provide a second indexed access method in the form of R-trees [15]). Indexing is also limited in that that an index can be created only on table columns whose data types are understood by the access methods and that an indexed scan of a table can exploit only those predicates that are understood by the access methods. For example, when B-tree is the sole indexed access method, only columns of built-in types can be indexed and only relational operators can be exploited during an indexed scan of a table. Several different approaches have been investigated to provide extensibility in access and indexing.

One approach is to support extensible indexing through query rewriting. Normally a separate entity manages the special access methods and indexing on complex data which the database engine is unaware of. A user query involving predicates on complex data is transformed into a different query by taking into consideration the access methods and indexing specific to complex data. The indices may be stored in a relational database or in external files. For example, in a

typical geographical information system (GIS) such as ESRI's SDE [10] a spatial data engine supplies a set of proprietary spatial functions and predicates and uses a relational database to store both user tables and *side tables* representing the spatial indices. Spatial predicates in a user query are converted by the spatial data engine into joins with and predicates on the *side tables*. The resulting query is then given to the relational database for optimization and execution.

The query rewriting approach does not require modifications to the database engine, which is normally not an option for application developers. When the underlying database does not support indexing on complex data, the query rewriting approach offers an excellent solution to build advanced spatial applications with good performance. However, a tighter integration of spatial indexing with the database engine can provide even better performance. This calls for enhancing the database engine with extensible access and indexing of complex data.

A better approach to extensible access and indexing is through user-defined access methods and user-defined indexing that are tightly integrated with the database engine. Stonebraker [24, 25] introduced user-defined access methods by describing an interface for implementing a new access method. Users specify functions for opening a scan of an index, getting the next record in a scan, inserting a record, deleting a record, replacing a record and closing a scan. These functions are called by the database engine at appropriate places when executing a query plan.

User-defined access methods have the performance advantage due to the tight integration of access methods with the database engine. However, experiences have shown that it is extremely difficult for application developers to define a new access method. The reason is that a new access method has to interface with several low level components of a database engine, including lock manager for locking on index objects, log manager for recovery, and buffer manager for page management. Few people possess such an intimate knowledge of the internals of a database engine, other than the database developers themselves, to be able to write an access method effectively. Extensive changes to low level components of a database engine are always a risky proposition that is not taken lightly in the context of real world applications of a mature database product.

Researchers have extended the concept of "search" in a user-defined access method in the form of generalized search trees [16]. The notion of a search key is generalized to a user-defined predicate that holds for every datum below the key in a search tree. Users define six key methods that are invoked during the top-down search and insertion/deletion of generalized search trees. This has been further extended in [1] to allow more powerful searches such as nearest-neighbor and ranked search.

Indexing aims to provide efficient search and querying of data using some underlying indexed access method. The purpose of user-defined indexing is to extend indexing capabilities to data types and predicates that may not be directly supported by the underlying indexed access methods. In [24], Stonebraker introduced a mechanism (called *extended secondary indices* in [20]) that allows users to apply existing access methods such as B-trees to new data types. In the case of B-trees, an *operator class* can be defined that provides a list of user-defined operators and specifies the correspondence between the user-defined operators and the standard relational operators for B-trees. Users can specify an operator class when creating an index on a table column of a new data type.

The mechanism of extended secondary indices in [24] is generalized in [20] so that predicates that cannot be mapped neatly to comparison operators can also be used for indexed scan of a table. For instance, one may want to index on keywords occurring in titles of books. Their idea is to introduce another operator that is applied to values of a column to generate index key values, e.g., an operator that returns a list of keywords occurring in a given string. The result of the operator can be a list of values that can be compared. The introduction of this operator essentially provides a logical separation between values of a table column to be indexed, e.g. the title of a book in a table books, and the corresponding index keys, e.g., keywords occurring in the title of a book.

This paper focuses on user-defined indexing and presents a high level framework of indexing for user-defined types. It generalizes extended secondary indices in [24, 20] in two aspects. First, we provide user control over "search" in indexing that maps a predicate with search argument into search ranges used by an underlying access method for indexed scan of a table. Such mapping is no longer limited to a single search range based upon a relational operator. For a user-defined predicate, users can provide their own method of generating possibly multiple search ranges based upon the search arguments of the predicate. Second, we provide user control over the execution of possibly expensive user-defined predicates using a multi-stage procedure.

As a generalization of extended secondary indices in [24, 20], our framework allows users to concentrate on the semantics of applications and user-defined predicates without being concerned with low level details of locking, recovery, buffer management or balanced search tree updates. It is tightly integrated with the database engine. It enhances the value of the underlying access methods, built-in or user-defined, in a database system by supporting indexing on new data types and indexed scans using new predicates. The multi-stage evaluation of expensive predicates provides

an implementation framework for filtering with approximate predicates [23].

The proposed framework has been implemented in IBM DB2 Universal Database. Its generality, ease-of-use and performance advantage have been demonstrated in several different domains of applications including, among others, spatial databases [26] and indexing [8] on structured XML (Extensible Markup Language) documents [27].

The rest of this paper is organized as follows. Section 2 revisits the implicit assumptions that are "hard-wired" in existing database systems that make them inadequate for indexing on user-defined objects. Section 3 presents our high level framework for direct user control over indexing of user-defined types. Section 4 demonstrates its application in two different domains. Section 5 concludes the paper.

2 "Hard-Wired" Indexing

B-trees are arguably the most popular indexed access method in relational databases [3, 9]. Existing database systems are heavily "hard-wired" to support only B-tree indexing on primitive data with relational operators. This section reviews the implicit assumptions that have been made and examines the consequent limitations for indexing of user-defined types.

With B-trees often as the built-in indexed access method, most database systems support primitive indexing that is limited by the B-tree indexed access method. Indices are defined by specifying the table name, the set of index columns, the sort order, and the unique constraint of an index, e.g.,

```
CREATE TABLE employee(empno Char(6),
                      name Char(20),
                      title Char(20),
                      salary Integer);
CREATE INDEX salary_index on employee(salary ASC);
```

When a tuple is inserted into a table, the values of all index columns are concatenated to form the index key that is used to traverse the B-tree for the insertion of the new index entry. Similarly when a tuple is deleted, the index key is used to identify the index entry for deletion.

Once an index is created for a table, subsequent queries can take advantage of the indexed access path provided by the index:

```
SELECT name, salary
FROM employee
WHERE salary > 50000;
```

Existing database systems make several implicit assumptions in their indexing support due to the use of B-trees as the built-in indexed access method. First, an index is created on the values of table columns directly. The index key is the concatenation of the values of the index columns. Clearly this is not acceptable for user-defined objects, which can be large binary objects or text documents that are not appropriate as index key values. Even if all the index columns have built-in types, users may want to create an index on some values *derived* from the values of the index columns, e.g., compensation level based upon the salary or keywords in the title of a book [20].

Second, a total order is assumed over the domain of index key values. An indexing search is restricted by a *single* range of index key values. For example, the predicate *salary* $>$ *50000* maps trivially to the range (50000, ∞). This is not sufficient for a user-defined predicate that may bound a search in more than one dimension, e.g., within a certain distance from a specific location.

Third, for index exploitation, which exploits any available indices for efficient query execution, only simple predicates of relational operators are considered by query optimizers. In a spatial database, a predicate such as *distance(location, point(10,10))* $\leq$ *5* may limit the search space of *location* within the circle centered at *(10, 10)* and with radius *5*. Query compilers need to be able to recognize user-defined predicates and know how to derive the corresponding search space in order to exploit any index of user-defined types for efficient query execution.

3 High Level Indexing of User-Defined Types

Our framework of high level indexing of user-defined types sits on top of the underlying access methods in a database system. It provides direct user control over index maintenance, search key generation for user-defined predicates and efficient predicate execution through filtering. This section describes the main components of the framework and its implementation in IBM DB2.

3.1 Index Maintenance

Index maintenance deals with the update of an index when tuples are inserted, deleted, or updated in a table. Existing database systems often treat the value of an index column directly as its index key, i.e., the mapping from values of an index column to index keys is the trivial identity mapping. To untie index keys from the values of an index column, we allow a user-defined *key transform*. Given the value of an index column, the key transform returns one or more index key values. Therefore a key transform is in general a table function that returns a table as its result. Each row in the result table forms an index key.

The introduction of key transforms brings several fundamental benefits. First of all, the domain of index keys is logically separated from the domain of values for an index column. Since an index column can be of any user-defined type, its values may be large objects

(LOBs) or structurally rich text documents, among other things. It is impossible to store them directly in an index. Nevertheless, an index can still be created on them using index keys derived by the key transform.

Second, even if the values of an index column are all of built-in types, using index keys derived by a key transform can have some nice properties that are not satisfied by indexing on the index column values directly. For example, a high dimensional space can be mapped to a linear ordered space such that multidimensional clustering is preserved and reflected in the one-dimensional clustering [2, 19]. Distance preserving transformations have been successfully used to index high dimensional data in many applications, such as time sequences [12] and images [11]. In [4], a new indexing method was proposed for high dimensional data spaces that can be implemented through a mapping to a one dimensional space. Key transforms allow the implementation of these new indexing methods on top of existing access methods such as B-trees.

Third, from an abstract interpretation point of view, index keys can be viewed as abstractions of the corresponding values of index columns and are simpler and/or occupy less space [8]. For spatial applications, index keys often represent approximations of spatial objects, such as z-values in [21] or minimum bounding boxes (MBRs) in R-trees [15]. Depending upon the abstraction defined by the key transform, the more information that is stored in an index, the more filtering that can be done by indexed search, thus offering a tradeoff between indexing cost and search efficiency.

Fourth, a single value of an index column can be mapped to a number of index keys using a table function as a key transform. The relationship between values of an index column and index keys is no longer one-to-one, but many-to-many, e.g., z-transform [21] and keywords in a book title [20]. Different values of an index column can have the same index key, and one value of an index column can have multiple index keys associated with it.

The idea of key transforms has been explored in [20] for keyword searching in textual databases. We are using the same idea as one of the building blocks for our framework of high level indexing of user-defined types.

3.2 User-Defined Predicates

Existing database systems support simple predicates of relational operators for which the corresponding search ranges can be easily determined based on the operators and the bound arguments. To provide extensible indexing over user-defined objects, two issues have to be tackled. First, a user-defined type may or may not support the standard relational comparisons. Even if it does, these relationships may not translate directly to search ranges over index keys. In addition, users may want to search based upon application-specific predicates other than relational comparisons, such as *overlap* and *within* in spatial databases.

Second, a predicate or condition defined by a user can be an arbitrary condition representing some complicated relationship among different objects. When such a user-defined predicate is used to exploit any existing index for efficient query execution, the rich semantics of user-defined predicates requires sophisticated computation of the corresponding search ranges to be used for the indexed scan of table. This is an efficiency issue as the complete range of index keys is always a logical candidate for search.

Example 3.1 Consider the following example:

```
CREATE TABLE customer(name varchar(20),
                      id integer, ...,
                      xyloc location);
...
CREATE INDEX locationIdx on customer(xyloc);
...
SELECT * FROM customer
WHERE within(xyloc, circle(...));
```

The search region is the minimal rectangular box (called minimal bound box), containing the circle in the WHERE clause. To capture the search region accurately, one might use two B-tree indexes, one on the X coordinate and the other on the Y coordinate of a location, for executing the query. We believe that an extensible mechanism is needed to provide user control over how the search region is determined given an arbitrary search condition.

□

For extensible indexing with user-defined predicates, we want to represent the corresponding search region as closely as possible and introduce the concept of *search methods*. Each search method is a user-defined function that given a semantic relation over user-defined objects and one of its search patterns, returns a *set* of search keys.

A search method computes the set of search keys over which the possible search targets can be found. For the query in Example 3.1, a search method can be defined for the semantic relationship *within* where the first operand is the search target and the second operand is the search argument. Assuming that an index key is a fixed size grid cell intersecting with an object, the search method can return the minimal set of grid cells that covers the circle given in the search argument.

A search method in general is only an approximation for the semantic relation r in the sense that every search target participating in the relation r with search arguments must have an index key among those returned by the search method. For instance, every geometric object that is *within* the circle given in the search argument must have a grid cell that is in the set of search keys generated by the search method.

However, a search method may not be accurate in the sense that some objects with an index key among those returned by the search method may not satisfy r with the search arguments. In other words, it may produce false hits. Therefore it is necessary in general to evaluate r for every object that is found using the index keys from a search method.

3.3 Index Exploitation

Index exploitation is performed by query optimizers in order to utilize any index for efficient query execution. Traditionally query optimizers have been able to exploit only simple relational operators for indexing since the corresponding search range can be easily determined. For index exploitation with user-defined predicates, the query compiler must be able to recognize them and find the relevant search methods to use. The definition of a user-defined function is extended to specify whether it can be used as a predicate and if so, what search method to use when certain operands are search arguments. (See Section 3.4 for details of the syntax of predicate specifications.)

For the query in Example 3.1, suppose that `within` has been defined as a predicate that has an associated search method when the second operand is a search argument. The query compiler can choose an index scan over a table scan to retrieve records from table `customer` for two reasons. One is that there is an index on `xyloc` attribute. The other is that the query compiler recognizes that the second operand of `within` is bound and `within` is a predicate with a search method when the second operand is a search argument. The index scan will use the corresponding search method to generate a set of search keys, which represents the minimal set of grid cells covering the circle in the second operand. The set of search keys will be used by the underlying access method to retrieve the relevant records from table `customer`.

3.4 Implementation and Predicate Filtering

The high level framework of indexing of user-defined types has been implemented in IBM DB2. Besides index maintenance, user-defined predicates and index exploitation, the implementation also provides user control over multistage evaluation of user-defined predicates through filtering. This avoids the potentially expensive evaluation of user-defined predicates and reduces both I/O and CPU costs.

Figure 1 shows the syntax for index extensions with the associated key transform and search methods. The semantic relation corresponding to a search method is not explicitly specified. The CREATE INDEX EXTENSION statement defines a parametric index extension. A parametric index extension is instantiated when an index is created on a table using CREATE INDEX statements. The parameters of an index extension can be used to specify, for example, the number of layers and the size of a grid cell in a multi-layer grid index.

The key aspects of an index extension include the key transform function (indicated by <*key transform invocation*>) and the associated search methods. Each search method contains a search key producer function (indicated by <*search key producer*>) that computes the set of search keys given search arguments and an index filter function (indicated by <*index filter*>) used inside the index component.

The user control over the index filter is a powerful concept. It provides early filtering using the index keys. This avoids the I/O cost of retrieving data that obviously do not satisfy the search criteria since data will not be retrieved from the disk using an index scan until the index keys are determined. This also makes it possible for users to combine multiple indexing mechanisms in a single search by plugging an index filter that performs additional search, e.g., using an external search engine.

Figure 2 shows the syntax of user-defined functions that can serve as predicates. Each predicate specification indicates an optional filter function, and the associated search methods for different search patterns. The data filter aims to reduce the potentially expensive evaluation of the predicate by filtering out records that do not satisfy the predicate using simpler and cheaper operation. In <*exploitation rule*>, the parameters following WHEN KEY indicate the search argument.

The optional keyword EXACTLY following AS PREDICATE requires a little explanation. When an index scan using a predicate is executed, the corresponding search method, which is a user-defined function, is invoked. It computes a set of search keys for the search target using the search arguments. The search keys are sent to the underlying access methods to retrieve the relevant records. The index filter associated with the search method is applied, if there exists one, before the records are retrieved from the disk. The relational data manager then applies the data filter associated with the predicate specification. Finally all records that pass through the data filter are evaluated using the predicate.

When the index filter and the data filter provide only an approximation to the predicate, e.g., in spatial applications, the final step of predicate evaluation is necessary. However, in other applications such as document search, the filters may compute *exactly* the set of all answers that satisfy the predicate. The final step of predicate evaluation should not be carried out in this case. The keyword EXACTLY indicates such a situation.

Figure 3 shows the architecture of the implementation in DB2. It can leverage any underlying access methods that are available. The rectangular boxes represent places where user-defined functions can be

```
<create index extension> ::=
        CREATE INDEX EXTENSION <header> <index maintenance> <index search>
<header> ::= <indexExtensionName> ( { <parmName> <parmType> }+ )
<index maintenance> ::=
        WITH INDEX KEYS FOR ( { <colName> <colType> }+ ) /* index columns */
        GENERATED BY <key transform>
<index search> ::=
        WITH SEARCH METHODS FOR INDEX KEYS
        ( { <colName> <colType> }+ ) {<search method>}+
<search method> ::=
        WHEN <searchmethodName> USING ( { <colName> <colType> }+ ) /* search arguments */
        RANGE THROUGH <search key producer>
        CHECK WITH <index filter>


<create index> ::=
        CREATE [UNIQUE] INDEX <indexName> ON <tableName>
        ( { <colName> [ASC | DESC ] }+ )
        USING <indexExtensionName> ( { <constant> }+ )
```

Figure 1: Syntax for index extensions, where N^+ specifies one or more occurrence of N, with the separator ',' when appropriate.

```
<create function> ::=
        CREATE FUNCTION <functionName> { <parmName> ] <dataType> }+
        <predicate specification>+
<predicate specification> ::=
        AS PREDICATE [EXACTLY]
        [ FILTER BY <data filter> ]
        [ <index exploitation> ]
<index exploitation> ::=
        SEARCH BY INDEX EXTENSION <indexExtensionName> <exploitation rule>*
<exploitation rule> ::=
        WHEN KEY ( { <paramName> }+ ) /* search target */
        USE <searchmethodName> ( { <paramName> }+ )
```

Figure 2: Syntax for user-defined predicates and their associated search methods

plugged in to support user-defined search.

The key transform is invoked in the index manager for index maintenance, when tuples are inserted/deleted/updates in a table. The query compiler utilizes specifications of user-defined predicates for index exploitation. During a search based upon a user-defined predicate, the corresponding search method is invoked by the relational data manager to generate a set of search keys.

For retrieval based upon a user-defined predicate, two filters are included in the architecture. The purpose is to avoid potentially expensive evaluation of user-defined predicates. Users can specify simpler and cheaper functions to be applied as filters before predicate evaluation. The index filter filters out records before they are retrieved from the disk into buffers inside the relational data manager. The data filter in the relational data manager presents another chance of cost efficient filtering before expensive predicates are evaluated.

4 Indexing for GIS Applications

In traditional geographical information systems (GISs), indexing on spatial data is provided through a set of proprietary APIs. When a query involves searching on spatial data, the spatial predicates are transformed for index exploitation, and the resulting query is then sent to the database for optimization and evaluation. The lack of integration of spatial indexing with the database engine leads to integrity issues and performance hits. Our framework of high level indexing makes it possible to have spatial indexing within a database and still take advantage of the special search methods that have been developed in GISs.

For example, suppose that the following user-defined types have been created:

```
CREATE TYPE envelop
AS (xmin int, ymin int, xmax int, ymax int);

CREATE TYPE shape AS (gtype varchar(20),
                      mbr envelop,
                      numpart sint,
                      numpoint sint,
                      geometry BLOB(1M))
                      NOT INSTANTIABLE;

CREATE TYPE point UNDER shape;
CREATE TYPE line UNDER shape;
CREATE TYPE polygon UNDER shape;
```

where `shape` serves a supertype for various subtypes

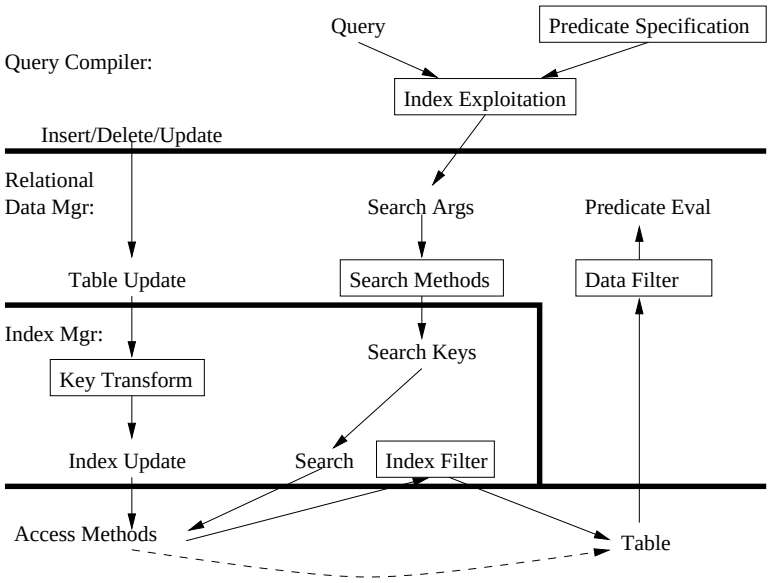

Figure 3: Implementation of high level indexing of user-defined types

such as lines and polygons. Two tables have been defined in the database, one storing the information about schools and the other containing the information on households and their income information.

```
CREATE TABLE schools AS (name varchar(20),
                         district varchar(20),
                         address varchar(20),
                         area shape
                         PRIMARY KEY (name, district));
CREATE TABLE households AS (address varchar(20),
                            annualincome int,
                            location shape);
```

The following query tries to compute the average annual income of all households inside the attendance area of a specific school:

```
SELECT avg(h.annualincome)
FROM houses h, schools s
WHERE s.name = 'Armstrong Elementary' AND
      s.district = 'Highland Park' AND
      within(h.location, s.area);
```

To allow efficient execution of this query, we need to (a) create an index extension incorporating user-defined key transform and search methods for `shape`; (b) create an index on table `households` using the index extension; and (c) specify predicates for `within` and the associated search methods.

The following statement defines an index extension over type `shape`. It uses a multilayer grid index for shapes.

```
CREATE INDEX EXTENSION
       gridshape(levels varchar(20) FOR BIT DATA)
WITH INDEX KEYS for (sh shape)
GENERATED BY gridkeys(
       levels, sh..mbr..xmin, sh..mbr..ymin,
               sh..mbr..xmax, sh..mbr..ymax)
WITH SEARCH METHODS FOR INDEX KEYS
       (level int, gx int, gy int,
            xmin int, ymin int, xmax int, ymax int)
  WHEN search_within USING (area shape)
       RANGE THROUGH gridrange(
             levels, area..mbr..xmin, area..mbr..ymin,
                     area..mbr..xmax, area..mbr..ymax)
       CHECK WITH checkduplicate(
             level, gx, gy, xmin, ymin, xmax, ymax,
             levels, area..mbr..xmin, area..mbr..ymin,
                     area..mbr..xmax, area..mbr..ymax)
  WHEN search_contain USING (loc shape)
       RANGE THROUGH gridrange(
             levels, loc..mbr..xmin, loc..mbr..ymin,
                     loc..mbr..xmax, loc..mbr..ymax)
       CHECK WITH mbroverlap(
             xmin, ymin, xmax, ymax,
             loc..mbr..xmin, loc..mbr..ymin,
             loc..mbr..xmax, loc..mbr..ymax);
```

The index extension definition specifies the function for key transform, `gridkeys` and two search methods. (DB2 uses the double dot notation for accessing attributes of objects of user-defined types.) One is for searching within a specific area, and the other is for finding shapes that contain a specific location. Both search methods use the same function, `gridrange`, to generate a set of index keys for potential search targets. Each search method has its own filtering function. All the functions that are mentioned may be user-defined functions, whose definitions are omitted here.

We are now ready to create an index on the `location` column of table `households`:

```
CREATE INDEX houselocIDX ON households(location)
             USING gridshape('10 100 1000');
```

The parameter indicates three levels of different grid cell sizes.

For index exploitation, we need to define predicates and their associated search methods. The following specification indicates that `within` should be viewed as a predicate.

```
CREATE FUNCTION within(s1 shape, s2 shape)
                RETURNS int
LANGUAGE C ... EXTERNAL NAME '/lib/gislib!within'
AS PREDICATE
FILTER BY mbrwithin(s1..mbr..xmin, s1..mbr..ymin,
                    s1..mbr..xmax, s1..mbr..ymax,
                    s2..mbr..xmin, s2..mbr..ymin,
                    s2..mbr..xmax, s2..mbr..ymax)
SEARCH BY INDEX EXTENSION gridshape
   WHEN KEY (s1) USE search_within(s2)
   WHEN KEY (s2) USE search_contain(s1);
```

The last three lines indicate that searching based upon predicates of `within` will be done using an index extension `gridshape`. When the first argument `s1` is the search target, use search method `search_within` with `s2` as the search argument. When the second argument `s2` is the search target, use search method `search_contain`. The query compiler is able to generate a plan that takes advantage of the access path provided by the index on `location` of table `households`. The key transform, search key producer, and filtering functions will be called automatically at appropriate places.

5 Performance

Our framework of high level indexing of user-defined types extends the expressive power and integrated optimization of SQL queries to user-defined types. This section presents some preliminary performance measurements for GIS applications using the existing GIS architecture and our integrated approach.

The existing GIS architecture is represented by SDE 3.0.2 on DB2 UDB Version 5 from ESRI [10], which uses a spatial data engine external to the database for spatial optimization. Given a table with business data and a column of spatial attributes, SDE introduces a new *feature table* to represent spatial data and a new index to process spatial queries. The feature table contains an id column as the primary key and all the spatial attributes and the geometric shapes. The spatial column in the original table (called *business table*) is replaced by an id column that is a foreign key for the feature table.

In addition to the feature table, SDE maintains a spatial index table, which uses a three level grid based index method in our example. The spatial index table contains the feature id (which is a foreign key for the feature table) and the indexing information such as the location of the lower left grid cell and the feature's minimum bounding rectangle (MBR).

When processing a spatial search query, SDE uses the spatial index table and the feature table to compute a list of (ids of) candidate shapes that satisfy the spatial predicate. The computed list of candidate shapes is then used to retrieve data from the business table by applying the remaining predicates in the WHERE clause of the spatial search query. Currently SDE handles the join between the business table and the feature table itself by executing different queries.

Our integrated approach of high level indexing of user-defined types is implemented in DB2 Spatial Extender.

We use the census block data for the state of Kentucky, which has 137173 polygons, with an average of 31 points per polygon. The table `kentuckyBlocks` has a column `boundary` of spatial type `POLYGON`, in addition to other attributes such as the name and the total population. Each polygon represents an area, containing as few as 4 points and as many as 3416 points.

```
CREATE TABLE kentuckyBlocks
       (name varchar(20), ..., boundary POLYGON)
```

The following queries represent some typical operations in GIS applications:

- *loading*: including raw data loading through a sequence of SQL insert statements and the maintenance of spatial indices;
- *region queries*: for three predefined regions in different locations, with the sizes of the answer sets being 3155, 2387 and 1457 respectively;
- *point queries*: 100 random point searches, simulating users pointing at a polygon during spatial browsing;
- *region queries with attributes*: same as *region queries* except that non-spatial attributes such as the name and the total population are also fetched in addition to the spatial data.
- *fetch all*: measuring how fast data can be pumped out of a database.

All queries were run on the IBM RS6000/J40 server and during off hours to minimize variations due to other users and processes. The GIS client programs are run on the same machine as the server. The measurements of query execution time (rounded to seconds) are shown in Table 1. Data loading was run once while the rest of the queries are run 3 times and the averages are shown.

In both *loading* and *fetch all*, we are processing the entire table and the integrated approach is about 4 times faster. In the case of *loading*, an insert statement for a row in the integrated approach becomes three insert statements in the GIS approach, one for the business table, one for the feature table, and one for the spatial index table. In the case of *fetch all*, since the GIS approach handles the join between the business table and the feature table by itself, it is executing a separate query against the feature table repeatedly, once for each set of data retrieved from the business table.

For *region queries* without non-spatial attributes, the integrated approach is about 2.5 times faster than the GIS approach, but is about 3 times faster for *region*

queries with non-spatial data. The difference is that the latter case involves the access of the business table. The GIS approach performs very well for *point queries*.

Overall, the results show that our integrated approach of high level indexing of spatial data has a much better performance. This shows the value of enhancing the database engine for extensible indexing of complex data.

6 Related Work and Conclusion

Universal applications involving both complex queries and complex data demand strong and flexible indexing support on non-traditional data such as geographical information and structured documents. Indexing of user-defined types with user-defined predicates is crucial to meeting the demands of modern database applications.

Different approaches make different tradeoffs when it comes to implementing indexing of user-defined types. The query rewriting approach transforms queries involving user-defined predicates into joins with special index tables that the database engine is not aware of. It does not require modification to the database engine, but at the same time, the database engine will not be able to take full advantage of the special indexing for query optimization.

One can also implement application-specific access methods. There is no shortage of special access methods for spatial or multidimensional data [13]. Generalized search trees have also been developed [1, 16] for user-defined access methods. They have the advantage of providing direct support for application-specific searches. Unfortunately, only B-trees [3, 9] and R-trees [15] have found their way into commercial database systems. One of the reasons is that an implementation of a new access method or a generic search tree is a huge undertaking since it interacts closely with low level components of the database engine such as concurrency control, lock manager and buffer manager. Reliability is of paramount importance for a mature database product, which often discourages extensive changes to low level components of the database engine. In addition applications are requiring new datatypes and more advanced searches like nearest neighbor for spatial data [22] or regular path expressions for semi-structured data [14]. It is expected that the access methods supported by a database system will not always match the growing needs of applications.

Our framework of high level indexing of user-defined types generalizes extended secondary indices in [24, 20]. It is tightly integrated with the database engine, especially with the index manager and query optimizer. It is orthogonal to the underlying access methods and can take advantage of any special access methods whenever they are available. Our main contribution is not in developing a new access method or a special search algorithm, but rather in providing a framework in which users have direct control over index maintenance, index exploitation, index filtering and predicate evaluation.

More specifically, users can define their own key transforms. The idea of key transforms is not new, e.g., transforming a geometric object into an MBR for R-trees [15] or into a set of z-values [21]. Following [24, 20], we give the power to users to decide what abstractions or approximations to use as index keys for a user-defined type.

Users can define their own search key producers for different search patterns of arbitrary predicates. Although search key producers are not sufficient by themselves to support advanced searches such as ranked and nearest neighbor (which require direct support from the underlying access methods), they bridge the semantic gap between user-defined predicates and the limited access methods that are available.

Users can define their own filters to avoid expensive predicate evaluation. Multistage predicate evaluation has been explored in [5, 6]. Researchers have also investigated query optimization issues with expensive predicates [7, 17] and with filtering using approximate predicates [23]. Our contribution is in integrating multistage evaluation of predicates with the database engine, especially the index manager, thus providing an implementation framework where approximate predicates can be utilized effectively for efficient query execution. As we have shown, the index filter is a powerful technique that makes it possible to avoid the I/O cost of retrieving useless data into the memory buffer. Furthermore, it offers an interesting mechanism to combine multiple indexing mechanisms in a single search, e.g., structured search with external full-text indexing.

The tight integration with the database engine means that it is possible for query compiler to exploit user-defined predicates in the *standard* framework of query optimization. This means that the full querying capabilities of SQL, including multiple predicates in a WHERE clause, aggregate functions, subqueries and recursion, are now available for universal applications through DB2.

References

[1] P.M. Aoki. Generalizing "search" in generalized search trees. In *IEEE Intl. Conference on Data Engineering*, pages 380–389, 1998.

[2] R. Bayer. The universal B-tree for multidimensional indexing. Technical Report I9639, Technische Universität München, Munich, 1996.

[3] R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

Queries	Loading	Region Queries			Region Queries w/Attr			Point Queries	Fetch All
		R1	R2	R3	R1	R2	R3		
GIS	3012	19	14	8	20	15	9	10	731
Integrated	706	8	5	3	8	5	3	8	170

Table 1: Performance measurements of spatial queries

[4] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *ACM SIGMOD Conference on Management of Data*, pages 142–153, 1998.

[5] T. Brinkhoff, H.-P. Kriegel, and R. Schneider. Comparisons of approximations of complex objects used for approximation-based query processing in spatial database systems. In *IEEE Intl. Conference on Data Engineering*, pages 40–49, 1993.

[6] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *ACM SIGMOD Conference on Management of Data*, pages 197–208, 1994.

[7] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Intl. Conference on Very Large Data Bases*, pages 87–98, 1996.

[8] J.-H. Chow, J. Cheng, D. Chang, and J. Xu. Index design for structured documents based on abstraction. In *Proceedings of the 6th International Conference on Database Systems for Advanced Applications*, pages 89–98, April 1999.

[9] D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 2(11):121–137, 1979.

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

[11] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994.

[12] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *ACM SIGMOD Conference on Management of Data*, pages 419–429, May 1994.

[13] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[14] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Intl. Conference on Very Large Data Bases*, pages 436–445, 1997.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD Conference on Management of Data*, pages 47–57, 1984.

[16] J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Intl. Conference on Very Large Data Bases*, pages 562–573, 1995.

[17] J.M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *ACM SIGMOD Conference on Management of Data*, pages 267–276, May 1993.

[18] Informix. Informix DataBlade Products, 1997. http://www.informix.com.

[19] H.V. Jagadish. Linear clustering of objects with multiple attributes. In *ACM SIGMOD Conference on Management of Data*, pages 332–342, May 1990.

[20] C.A. Lynch and M. Stonebraker. Extended user-defined indexing with application to textual databases. In *Intl. Conference on Very Large Data Bases*, pages 306–317, 1988.

[21] J.A. Orenstein and F. Manola. PROBE spatial data modeling and query processing in an image database applications. *IEEE Transactions on Software Engineering*, 14(5):611–629, May 1988.

[22] N. Roussopoulos, Kelley S., and F. Vincent. Nearest neighbor queries. In *ACM SIGMOD Conference on Management of Data*, pages 71–79, 1995.

[23] N. Shivakumar, H. Garcia-Molina, and C.S. Chekuri. Filtering with approximate predicates. In *Intl. Conference on Very Large Data Bases*, pages 263–274, 1998.

[24] M. Stonebraker. Inclusion of new types in relational data base systems. In *IEEE Intl. Conference on Data Engineering*, pages 262–269, February 1986.

[25] M. Stonebraker and P. Brown. *Object-Relational DBMSs: Tracking the Next Great Wave.* Morgan Kaufmann Publishers, Inc., 1999.

[26] Y. Wang, G. Fuh, J.-H. Chow, J. Grandbois, N.M. Mattos, and B. Tran. An extensible architecture for supporting spatial data in RDBMS. In *Proceedings of Workshop on Software Engineering and Database Systems*, 1998.

[27] *Extensible Markup Language (XML)*, 1997. http://www.w3.org/TR/WD-xml-lang.