Developing a DataBlade for a New Index

Rasa Bliujūtė Simonas Šaltenis Giedrius Slivinskas Christian S. Jensen
Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7E, 9220 Aalborg, Denmark
{rasa, simas, giedrius, csj}@cs.auc.dk

Abstract

In order to better support current and new applications, the major DBMS vendors are stepping beyond uninterpreted binary large objects, termed BLOBs, and are beginning to offer extensibility features that allow external developers to extend the DBMS with, e.g., their own data types and accompanying access methods. Existing solutions include DB2 extenders, Informix DataBlades, and Oracle cartridges. Extensible systems offer new and exciting opportunities for researchers and third-party developers alike. This paper reports on an implementation of an Informix DataBlade for the GR-tree, a new R-tree based index. This effort represents a stress test of the perhaps currently most extensible DBMS, in that the new DataBlade aims to achieve better performance, not just to add functionality. The paper provides guidelines for how to create an access method DataBlade, describes the sometimes surprising challenges that must be negotiated during DataBlade development, and evaluates the extensibility of the Informix Dynamic Server.

1. Introduction

Advanced applications continuously emerge that pose new requirements to database management systems, including the need for efficient handling of the complex types of data inherent to geographical, multimedia, medical, and other advanced applications. Such data include images, videos, documents, as well as data with temporal and spatio-temporal references. Most relational DBMSs provide binary large objects, which may be used for storing such data, but this is generally not satisfactory because the internal structure of data is invisible to the DBMS, which then cannot provide efficient access to the data. Support for new data types can also be introduced at the application level. But this does not provide efficient access, and it is also not economic for the many applications that need similar support to reimplement similar ad-hoc solutions.

New complex data types, including efficient querying capabilities on them, should be supported by the DBMS. Because new applications will continue to appear that require support for new kinds of data, the DBMS should be extensible, allowing the users themselves to extend the DBMS's functionality. This alleviates the vendors from attempting to keep up with the demands for new data types, and it allows users to obtain support for very specific kinds of data, for which there is only a very small market; the vendors have little incentive to develop support for such data.

Indeed, over the last couple of years, major DBMS vendors have come up with new technology that allows the users themselves to extend the DBMS's functionality. Examples include DB2 *extenders*, Informix *DataBlades*, and Oracle *cartridges*. Extenders, DataBlades, and cartridges can be developed separately and plugged into the appropriate DBMS.

This technology allows application developers to add new functionality to a DBMS according to their concrete needs, as well as gives third-party vendors an opportunity to make products targeting a specific application area. In addition, extensible database technology reduces the gap between real products and new techniques proposed by the research community, because these techniques can be integrated into DBMSs more easily. This facilitates dissemination of research results and the transition from research results to products.

The paper describes a prototype implementation of a new access method, termed the GR-tree [4], as an Informix DataBlade. Based on the R*-tree [3] (an improved version of the R-tree originally proposed by Guttman [7]), this tree indexes now-relative bitemporal data, which is data with associated valid-time and transaction-time values [14]. Many real-world databases contain a significant portion of this type of data.

The paper reports the experiences gained from developing the DataBlade. It provides systematic guidelines for how to create an access method DataBlade, while also pointing out issues—expected as well as unexpected—that proved to be particularly challenging when building the DataBlade. Informix was chosen because it provides the possibility to add advanced user-defined data types as well as user-defined access methods for these new data types. The paper covers issues related to the design of the required new data type that accompanies the new access method; it discusses the design of the functions used internally in the access method and the functions that may appear in WHERE clauses of SQL queries and that trigger the use of the access method; and it addresses issues related to concurrency control and recovery. The coverage of the actual implementation effort encompasses the available development tools and the specific coding tasks.

The presentation is structured as follows. Section 2 explains what bitemporal data is and how it may be represented. Section 3 briefly describes the GR-tree. On this background, Section 4 presents the general steps needed to develop an access method DataBlade. Section 5 reports on specific challenges encountered when these steps were performed to create the GR-tree DataBlade, and Section 6 describes the implementation. Section 7 concludes and offers observations about Informix's applicability for the implementation of new access methods.

2. Bitemporal Data

In this section, we introduce bitemporal data, showing that the time associated with bitemporal data can be viewed as two-dimensional regions, which suggests that bitemporal data may be indexed using adapted spatial indices.

Two temporal aspects of database tuples, termed valid and transaction time, have proven to be of interest in a wide range of database applications. The valid time of data captures when the data is true in the modeled reality, while the transaction time is the time during which the data is current in the database [14]. These two aspects are orthogonal in that each could be independently recorded, and each has specific properties associated with it. The valid time of a tuple can be in the past or in the future (allowing a database to store information about the past and the future) and can be changed freely. In contrast, the transaction time of a tuple cannot extend beyond the current time and cannot be changed. Data having associated both valid and transaction time is termed bitemporal data. Bitemporal data is now-relative if the end of valid time or the end of transaction time is not fixed, but instead tracks the current time and continuously extends as time passes.

Table 1 exemplifies now-relative bitemporal data, which is represented using TQuel's four-timestamp format [20]. With this format, each tuple has four time attributes. Now-relative tuples are represented using UC (denoting "until changed") and NOW variables [6] for transaction- and valid-time end attributes, respectively. The time granularity is a month, and the current time is assumed to be 9/97.

Tuple (1) records that the information "John works in Advertising" was true from 3/97 to 5/97 and that this was recorded during 4/97 and is still current. Tuple (3) records that "Jane works in Sales" from 5/97 until the current time,

	Employee	Department	TT1	TT2	VT1	VT2
(1)	John	Advertising	4/97	UC	3/97	5/97
(2)	Tom	Management	3/97	7/97	6/97	8/97
(3)	Jane	Sales	5/97	UC	5/97	NOW
(4)	Julie	Sales	3/97	7/97	3/97	NOW
(5)	Julie	Sales	8/97	UC	3/97	7/97
(6)	Michelle	Management	5/97	UC	3/97	NOW

Table 1. The EmpDep Relation

that we recorded this belief on 5/97, and that this remains part of the current database state.

The temporal aspect of a tuple can be represented graphically by a two-dimensional ("bitemporal") region in the space spanned by valid and transaction time. Cases 1–5 in Figure 1 illustrate the *bitemporal regions* of Tuples (1–4) and (6), respectively.

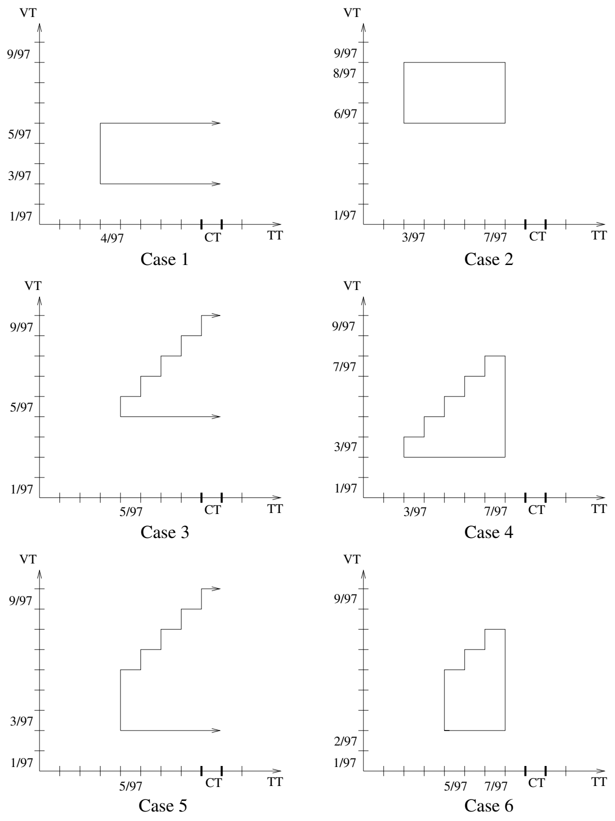


Figure 1. Bitemporal Regions

A now-relative transaction-time interval yields a rectangle that "grows" in the transaction time direction as time passes (Tuple (1), Case 1). Having both transaction- and valid-time intervals being now-relative yields a stair-shaped region growing in both transaction time and valid time as time passes (Tuple (3), Case 3). Information can be recorded in the database after it becomes true in the modeled reality. In this situation, having both the transaction- and valid-time intervals being now-relative yields a stair-shape with a high first step (Tuple (6), Case 5).

It is also possible to record information in the database before it becomes true in the modeled reality (Tuple (2), Case 2). If, at some time, a tuple is logically deleted (its transaction-time end value UC is changed to the fixed value 'current time'-1), the bitemporal region stops growing (Tuples (2), (4); Cases 2, 4, 6).

Two-dimensional bitemporal regions can be indexed using adapted spatial indices. An essential challenge in indexing bitemporal data is to properly handle now-relative intervals. The next section briefly presents the GR-tree [4] which contends well with this requirement and outperforms other indices for now-relative bitemporal data. A description of the implementation of the GR-tree as an Informix DataBlade follows next.

3. The GR-Tree Index

The GR-tree is based on the R*-tree [3], which is a spatial index consisting of nodes organized in a tree structure. A node contains a number of entries and is stored in one disk page. Spatial objects are bounded with minimum bounding rectangles, which are stored in leaf-node entries together with pointers to data tuples containing the spatial objects. All entries of each non-root node are also bounded with a minimum bounding rectangle, that, together with a pointer to the node, composes an entry of a parent node. Figure 2(a) shows minimum bounding rectangles of spatial objects.

When a query asking to retrieve all spatial objects that overlap with a given query region is issued, the tree is traversed down from the root looking for entries that encode rectangles overlapping with the bounding rectangle of the query region. The list of qualifying entries is obtained, and then the spatial objects are retrieved from the corresponding data tuples. The last step is to check using the exact geometry whether the query region actually overlaps with the retrieved spatial objects.

Because the R*-tree cannot handle the growing bitemporal regions presented in Section 2, it was modified, leading to the GR-tree [4]. Variables UC and NOW were introduced in node entries at all tree levels, making it possible to record the exact geometry and the temporal behavior of the bitemporal regions in leaf-node entries. Entries in non-leaf nodes store minimum bounding regions of the child nodes. These minimum bounding regions can be either rectangles or stair-shapes. Minimum bounding regions grow when the regions inside them grow. Figure 2(b) illustrates minimum bounding regions of the GR-tree. Note that node 2 is bounded with a stair-shape because none of its included regions extend above the line VT = TT.

The layout of a GR-tree node does not differ significantly from the layout of an R*-tree node. A leaf-node entry contains four timestamps encoding a bitemporal region and a pointer to the actual bitemporal data stored in the database.

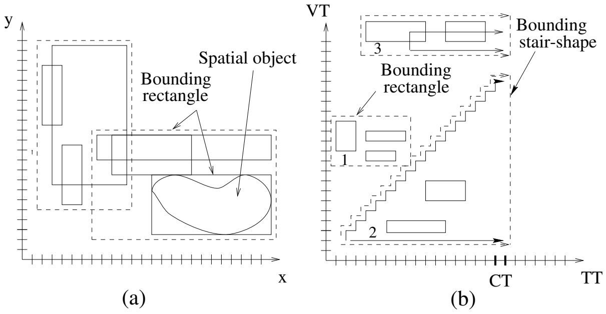


Figure 2. Graphical Representation of (a) R*-Tree and (b) GR-Tree

A non-leaf node entry contains four timestamps, a flag "`Rectangle`," a flag "`Hidden`," and a pointer to a child node. Here, timestamps represent a minimum bounding region that encloses all child-node regions. The "`Rectangle`" flag denotes whether timestamps of the form (tt1, UC, vt1, NOW) represent a stair-shape *or* a rectangle growing in both transaction and valid time. The "`Hidden`" flag is used to track growing stair-shapes that are placed in a larger bounding rectangle having a fixed valid-time end (that is bigger than the current time).

The algorithms to accompany the GR-tree structure are based on the R*-tree algorithms, which are modified to contend with growing regions encoded using the flags and timestamp variables. The GR-tree algorithms resolve variables UC and NOW according to the current time. New insertion algorithms that take into account the varying shapes of the bitemporal regions were designed for the GR-tree. Reference [4] contains in-depth descriptions of the algorithms and performance tests, showing that the GR-tree outperforms the other proposed indices for general bitemporal data.

4. The Steps Needed to Implement an Access Method DataBlade

In this section, we give guidelines for how to implement an access method DataBlade module in Informix Dynamic Server with Universal Data Option (the abbreviation "Informix Server" will be used throughout). Section 5 presents the GR-tree-specific design considerations, and Section 6 describes the implementation.

The GR-tree DataBlade was developed using the C and C++ programming languages, and using the DataBlade API [9], the Virtual-Table Interface API [12], and the Virtual-Index Interface API [11] of the Informix Server.

In Informix terms, a *secondary access method* is an index *type*, e.g., the B$^+$-tree. Meanwhile, a *virtual index* is a specific index *instance* of a *developer-defined* secondary access method. A developer can define a secondary access method

Task	Access Method Purpose Functions
Creating and dropping an index.	`am_create()`, `am_drop()`
Opening and closing an index.	`am_open()`, `am_close()`
Scanning an index for records that meet the qualifications of a query.	`am_beginscan()`, `am_endscan()`, `am_rescan()`, `am_getnext()`
Adding, deleting, and updating records in an index.	`am_insert()`, `am_delete()`, `am_update()`
Determining the cost for a scan of an index.	`am_scancost()`
Updating statistics.	`am_stats()`
Checking an index consistency.	`am_check()`

Table 2. Tasks of Access Method Purpose Functions

("access method" for short) by providing a set of functions that will be used by the Informix Server to access and manipulate instances of the access method, i.e., virtual indices. By creating a new access method, an alternative indexing strategy for specialized data can be provided.

Thus, to enable usage of the GR-tree in Informix, a GR-tree access method has to be created. Then, any number of GR-trees can be created using this access method. To accomplish this, a total of six steps, described below, must be completed. Steps 1–4 create an access method, and Steps 5–6 create a virtual index using the access method.

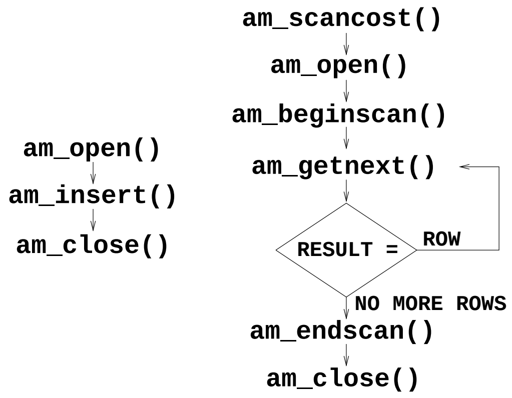
Step 1: Create new data types if needed.
Informix allows a DataBlade developer to define new data types to support new kinds of data. In addition, a developer can (1) write functions implementing operations, e.g., arithmetic or comparison, to be used on the new data type and (2) provide casts for data conversions between the new data types and existing data types. A discussion about the choice of data type for time extents in the GR-tree DataBlade is given in Section 5.1.

Step 2: Create access method purpose functions.
Access method purpose functions ("purpose functions" for short) manipulate an index structure. These functions are data-type independent and implement the skeleton of the access method; additional logic necessary for the data types that the access method is to support is added via operator classes (see Step 4). The purpose functions are to be coded in C/C++, compiled, and registered using the `CREATE FUNCTION` statement (the path and name of the file where the executable code of a function resides have to be known). The following example registers a purpose function which will be used in the GR-tree access method.

```
CREATE FUNCTION grt_open(pointer)
RETURNING int
EXTERNAL NAME "src/grtree.bld(grt_open)"
LANGUAGE c;
```

Table 2 lists and briefly describes the generic purpose functions that may be specified for an access method. Only the `am_getnext()` function is mandatory.

If the Informix Server determines that a table specified in an SQL statement should be accessed via a virtual index, it dynamically loads and executes the appropriate purpose functions. Figure 3 shows which purpose functions are called if the Informix Server determines that a virtual index should be used when processing `INSERT` and `SELECT` statements, respectively.



(a) (b)

Figure 3. Access Method Purpose Functions Called for (a) `INSERT` and (b) `SELECT`

A number of structures, termed *descriptors*, are used in the purpose functions. The descriptors contain the information that the purpose functions need to perform a scan, an insertion, an update, or a deletion in a virtual index. The Informix Server fills in most of the data of a descriptor and passes it to the purpose functions. For instance, when the Informix Server invokes `am_beginscan()`, it passes as an argument a so-called *scan descriptor*, which contains information about the qualification condition. Descriptors can also contain user-defined data. Data in the descriptors is accessed using specific functions.

Step 3: Register the access method.
The purpose functions have to be registered as part of the access method using the `CREATE SECONDARY ACCESS_METHOD` statement. An example of how to register the `grtree_am` access method is given below (value `"S"` for `am_sptype` means that virtual indices will be created in `sbspace`, see Section 5.3).

```
CREATE SECONDARY ACCESS_METHOD grtree_am
(am_create = grt_create,
 am_open = grt_open,
 am_getnext = grt_getnext,
 am_close = grt_close,
 am_drop = grt_drop,
 am_sptype = "S" );
```

Step 4: Create operator classes.
The purpose functions manipulate the index structure, but are not data-type specific. In contrast, an operator class is a set of functions that allows an access method to manipulate values of particular data types. An operator class consists of functions implementing those operations on the data types that are supported by the access method. In general, there can exist several operator classes for the same access method (see Figure 4), but normally one is enough. More are needed when a different access method behavior has to be specified; the situations where this can occur are considered below. An "off-the-shelf" access method DataBlade always contains at least one operator class.

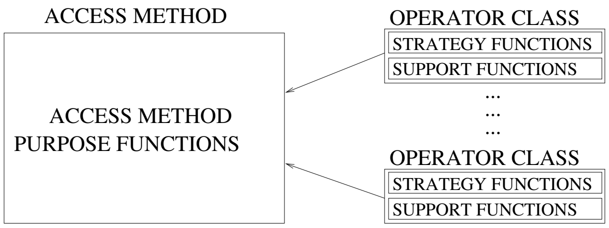


Figure 4. Association Between an Access Method and Operator Classes

Operator class functions have to be written, compiled, and registered as user-defined routines (UDRs) using the `CREATE FUNCTION` statement. These functions are divided into two categories: *strategy* and *support* functions. Strategy functions specify the interface between SQL and the access method. These boolean functions are typically used in WHERE clauses of SQL statements by the application programmer. An example strategy function for the R-tree access method is `Overlap()`. Functions `GreaterThan()` and `LessThanOrEqual()` are among the strategy functions of the B-tree operator class.

When the query optimizer meets a function in the WHERE clause of an SQL statement, it determines if a virtual index is applicable for the processing of the SQL statement by checking if a virtual index exists for the column involved in the function and if this function is declared as a strategy function in the operator class of the corresponding access method. When processing the SQL statement, the purpose functions are invoked as shown in Figure 3(b). Function `am_getnext()` dynamically resolves which strategy function is used and invokes that function on index entries to find the qualifying regions.

To enhance an existing access method with support for a new data type, new additional strategy functions with the same names, but new argument types, should be written and registered. This way, the existing operator class is extended. A new operator class must be created when there is a need to employ new strategy functions or to redefine the existing ones. For instance, creating a new operator class for the R-tree access method, the new strategy function `Neighbour()`—which finds all objects that are close to the query region, but do not overlap with it—can be added. When an existing operator class is extended or a new one is created, the purpose functions do not require any modifications.

Support functions are used only internally by the access method to maintain the index structure and are usually not invoked from SQL, but they are visible to the programmer since they are registered as UDRs and declared in the operator class of the access method. An example of a support function for the R-tree access method is `Intersection()`, which computes the intersection of two minimum bounding rectangles. In the same way as for strategy functions, the purpose functions dynamically resolve and invoke appropriate support functions. Support functions for new data types may be registered, extending an existing operator class; or redefined support functions can be employed registering a new operator class. The latter can be exemplified as follows. The B^+-tree operator class contains a support function `compare()`, which compares two values of several data types. The natural order for integers is -2, -1, 0, 1, 2, but the programmer may want to change this order to 0, -1, 1, -2, 2. Then a substitute function for `compare()` has to be written, and a new operator class with the new function name instead of the old one has to be registered for the B^+-tree.

Alternatively, support functions can be "hard-coded," i.e., they can be statically linked together with the purpose functions, not registered as UDRs and not declared in any operator class of the access method, but explicitly invoked from the purpose functions where appropriate. This way, the programmer does not know about their existence, cannot add support for new data types by extending the existing operator class, and cannot substitute the "hard-coded" support functions with the redefined ones via a new operator class. Similarly, strategy functions can also be "hard-coded" in the `am_getnext()` purpose function. Unlike "hard-coded" support functions, "hard-coded" strategy functions must still have corresponding registered UDRs so that these can be invoked when an SQL statement is processed without using a virtual index. These UDRs must also be declared in an operator class so that the optimizer knows when a virtual index can be used. But since the purpose functions explicitly invoke "hard-coded" strategy functions, the support for new data types cannot be added by extending the existing operator class, and new or redefined strategy functions cannot be employed via new operator classes.

In general, it depends on the specifics of an access method whether it makes sense to offer a future possibility to extend existing or create new operator classes. The cost of this extensibility is the overhead of dynamic resolution and execution of strategy and support functions. For general access methods, such extensibility may be a desirable option. If an access method is targeted for some specific data type and a specific set of strategy and support functions or if simpler and more efficient code is preferred, it may be more reasonable to internally "hard code" all function invocations.

An operator class for an access method is created using the `CREATE OPCLASS` statement. The following example shows how the operator class for the `grtree_am` can be created (the GR-tree operator class and its functions are described in Section 5.2).

```
CREATE OPCLASS grt_opclass FOR grtree_am
STRATEGIES(grt_overlap, grt_contains,
           grt_containedin, grt_equal)
SUPPORT(grt_union, grt_size,
        grt_intersection);
```

Step 5: Create storage space for a virtual index.
The space for a virtual index has to be created using the `onspaces` command, see Section 5.3.

Step 6: Create a virtual index.
A virtual index is created using the `CREATE INDEX` statement. When creating a virtual index on a single column or on a number of columns, the operator class has to be specified for each column. The following example shows how a virtual index is created in storage space `spc` using the `grtree_am` access method.

```
CREATE INDEX grt_index
       ON employees(column1 grt_opclass)
USING grtree_am
IN spc;
```

5. Design Considerations for Implementing the GR-Tree DataBlade

Implementing the GR-tree DataBlade according to the steps outlined in the previous section, a number of technical design issues had to be resolved; some solutions were not straightforward. This section reveals some of the hidden challenges that a DataBlade developer should be prepared to face. Section 5.1 discusses the choice of a new data type for now-relative bitemporal data. Section 5.2 presents the GR-tree operator class. Section 5.3 provides insights into the possible index storage options. Deletions and the handling of the database variables UC and NOW are discussed in the full version of the paper [5].

5.1. Physical Representation of a Time Extent

The GR-tree indexes the time extents associated with the database records. We have so far assumed that each of the four timestamps is in a separate column (see Section 2). In this section, we also consider other options for the implementation, i.e., we discuss what number of columns and what data types should be used in an Informix physical table to represent the time extents.

In general, three alternatives for representing time extents are natural: using four columns, two columns, or one column. Informix has a set of built-in data types and allows the user to construct new data types. The built-in data types `DATE` and `DATETIME` are suitable for representation of time. These would make it possible to store time extents of records in four columns of a table—one column for each of `TT1`, `TT2`, `VT1`, and `VT2`—and values in these columns would be of type `DATE` or `DATETIME`. But, since we want to use the special values UC and NOW for `TT2` and `VT2`, respectively, the built-in data types are not suitable. To be able to interpret UC and NOW, a new date type must be constructed, which would not be interpreted by Informix. Functions interpreting this type must be provided. This kind of data type is termed an *opaque* data type.

As an alternative to having four columns for a time extent, two columns could be used: one representing an interval of valid time and another representing an interval of transaction time. Yet another possibility would be to have only one column, completely representing the time extent of a record. As for the four-column alternative, opaque types have to be constructed for these two alternatives. This could be done in two ways. One way is to construct an opaque *Date* type supporting values UC and NOW, and then to use a *collection* data type with two (for intervals) or with four (for a whole time extent) members of the opaque *Date* type. Another way is to directly construct an opaque type *Interval* or *Extent*, not using a collection data type.

Issues related to the specifics of querying time extents and declaring operator class strategy functions affect the choice of how to represent time extents.

In order to correctly decode a time extent of a tuple, all its four time values must be interpreted together. This restricts the design alternatives. To illustrate, consider the record in Table 3, whose corresponding bitemporal region is shown in Figure 5. Consider the query "Who worked in the Sales department during 7/97 according to the knowledge we had during 5/97?" issued at the current time, 9/97. If the valid- and transaction-time intervals are considered separately when answering this query, the answer will include Julie. But this would be incorrect, because `VT2` is NOW and Julie's bitemporal extent is a stair-shape, which does not overlap with the given query region. Our "bitemporal" function cannot be replaced by two functions that consider transaction- and valid-time intervals separately.

Name	Department	TT1	TT2	VT1	VT2
Julie	Sales	3/97	7/97	3/97	NOW

Table 3. The EmpDep Relation

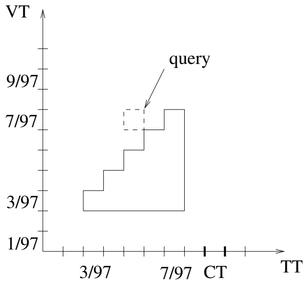


Figure 5. Time Extent of the Julie Record

Functions registered with the `CREATE FUNCTION` statement can be used in SQL statements. If a function is also declared as a strategy function in an operator class of an access method, an index can be used (if it exists) processing the SQL statement involving that function. If the index is used, the Informix Server passes the relevant part of the WHERE clause to the index interface in a special structure called a *qualification descriptor* (which is a part of the scan descriptor). This structure is restricted to accommodate only single-column predicates, which implies that only single-column functions can be declared as strategy functions, i.e., can be supported by a virtual index.

According to this, to enable virtual index usage processing an SQL statement with a WHERE clause including a bitemporal function (requiring four time extent values), this function must be a single-column function. A time extent of a record thus cannot be represented using four or two columns, so we represent it as one column, and the values in this column are of our newly created opaque data type, `GRT_TimeExtent_t`.

The last issue to consider about date types in the GR-tree is the time granularity. For our prototype implementation, we chose a granularity of days, as provided by the `DATE` type. Many other possibilities exist, one being fractions of a second, as provided by the `DATETIME` type.

5.2. The GR-tree Operator Class

In the default operator class of the R-tree access method, the strategy functions include `Overlap()`, `Equal()`, `Contains()`, and `Within()`, and the support functions include `Union()`, `Size()`, and `Inter()`. When the query optimizer examines a WHERE clause that involves some function on some column, it checks if this column has an R-tree defined on it. If it does, and if the function is one of the strategy functions of the operator class of the R-tree access method, the optimizer can choose to use the R-tree to process the SQL statement (see Section 4).

In a similar manner, strategy functions in the GR-tree access method operator class are `Overlaps()`, `Equal()`, `Contains()`, and `ContainedIn()`. Having a table `Employees` with columns `Name`, `Department`, and `Time_Extent` and having a GR-tree index on the `Time_Extent` column, consider the following query.

```
SELECT Name
FROM Employees
WHERE Overlaps(Time_Extent,
               "12/10/95, UC, 12/10/95, NOW")
```

Function `Overlaps()` takes two `GRT_TimeExtent_t` objects as input and returns a boolean value. The Informix Server examines the WHERE clause of this query as described above and decides whether or not to use the index. If the index is not used, `Overlaps()` is invoked for each table record. If the index is used, the function must be invoked traversing the index from the root, looking for entries that have regions overlapping with the region specified by `"12/10/95, UC, 12/10/95, NOW"`.

Recall that the four timestamps in an internal-node entry do not uniquely identify the shape of the region (see Section 3). This means that another function (let us call it `OverlapsInternal()`) should be used to determine whether the given region overlaps with a region encoded by an internal-node entry. If `OverlapsInternal()` is registered as a UDR, a type for internal-node regions should also be registered as an opaque type. In the following, we discuss three alternatives for designing the GR-tree operator class. The alternatives differ in the extensibility they provide and in the simplicity and efficiency of the access-method code.

The first and simplest alternative is not to create a new opaque type for internal-node regions. To avoid this, all strategy and support functions that take as arguments or return as results internal-node regions must be “hard coded.” Hard coding makes it impossible to extend the existing operator class or to create new ones. For example, if we do not register a new opaque type and hard code `OverlapsInternal()`, a new operator class, where `Overlaps()` is replaced by a new strategy function, cannot be defined. Implementing a new version of `Overlaps()` would mean that a new `OverlapsInternal()` should be implemented, and this cannot be done, because the old hard-coded version of `OverlapsInternal()` is called from the purpose functions.

The second alternative is to create a new opaque data type for internal-node regions. Then a function operating on internal-node regions (such as `OverlapsInternal()`) could be registered as a UDR under the same name as the corresponding function operating with `GRT_TimeExtent_t` objects, e.g., `Overlaps()`. Following this approach, implementing a new strategy function would mean additionally implementing its ”internal” function for internal-node regions. This way, the operator class would be extended for internal-node regions.

Yet another approach would be to have a single type `GRT_TimeExtent_t` that would encode both the internal-node regions and the bitemporal regions stored in database tables and leaf-nodes of an index. To use space efficiently, this opaque data type can be declared to have a variable length. The main requirement is that the `Overlaps()` function should be able to determine whether its arguments are real bitemporal regions or internal-node regions. This approach is followed by Informix in its R-tree access method implementation.

The latter two approaches make it possible to extend the existing operator class and to create new ones with new or redefined functions. The cost for such extensibility is the overhead of dynamic resolution and execution of strategy and support functions. Since the GR-tree is an access method for a quite specific data type, the hard-coding approach is used in the GR-tree DataBlade implementation.

One of the shortcomings of Informix's operator class framework and the UDR framework in general is that there are rather limited means of telling the query optimizer about associations between UDRs. To illustrate this, consider this situation: the GR-tree operator class contains a strategy function `Overlap()`, but does not contain a strategy function `Equal()`. If a user asks in a query for regions that are equal with a given region, the virtual index will not be used. However, the optimizer could use the index to find all regions that overlap with the given region and then check whether or not they are equal (significantly less regions will have to be retrieved from disk). But there is no way of telling the optimizer that if two regions do not overlap, they cannot be equal, because Informix only allows to specify that one function is a negator of another function (returns the opposite, given the same set of values, e.g., `Equal()` and `NotEqual()`) or that one function is a commutator of another function (returns the same result when its arguments are passed in reverse order, e.g., `GreaterThan()` and `LessThanOrEqual()`).

5.3. Storage Options, Concurrency, and Recovery

In this section, we consider the possibilities available for index storage. The choice of storage mechanism has important implications for concurrency control and recovery.

A developer of an access method DataBlade has two options for the storage of an index. One possibility is to store index data outside the Informix data space, e.g., in a regular operating system file. Another possibility is to store the index as one or several large objects, in a so-called smart-blob space, abbreviated *sbspace*. Table data as well as data for built-in access methods such as B^+-trees or R-trees are stored in *dbspaces* in the Informix Server. We do not consider this latter option for an access-method DataBlade because there is no public interface for accessing dbspaces. Before investigating each of the two available options, we first briefly recall the proposals for concurrency control and recovery in tree-based index structures.

The simplest solution to concurrency problems in a tree-based index structure is to lock the entire tree or the subtree that needs to be modified. The upper levels of the subtree are locked so that only readers can access them [2]. To achieve much higher levels of concurrency, B-link trees [18] and R-link trees [15] were proposed, and Kornacker et al. [16] generalized the ideas of R-link trees to apply to a broader class of tree-based access methods. Using the link-based concurrency control protocols, a lock on a parent node can be released before visiting a child node. To use this protocol in an access-method DataBlade, either the Informix Server or the DataBlade developer must provide the full management of locks on the tree nodes. The recovery protocol proposed by Kornacker et al. in addition calls for special types of log records and a logical undo capability.

Informix's own predefined R-tree access method stores its indices in dbspaces, the Informix page manager provides the appropriate concurrency control, and the Informix log manager provides the appropriate recovery mechanisms. Thus, the R-tree access method can implement the above-mentioned R-link mechanism.

For sbspaces, Informix provides automatic two-phase locking at the large-object level. Locks are acquired upon opening a large object for reading or writing and, depending on the lock mode and the isolation level of a transaction, are released either upon closing the object or at the end of a transaction. The DataBlade developer may vary the number of large objects used for storing index data. Possibilities range from storing the whole index in one large object, having the least possible concurrency, to storing each index node in a separate large object. The latter option has the serious drawback that the large-object handles that should be stored in the internal nodes of a tree (as pointers to child nodes) are relatively large. In addition, opening and closing large objects can be time consuming. It may be worth investigating design options in-between these two extremes, where large objects do not store single nodes, but several nodes, e.g., subtrees. Such a design would require policies for assigning nodes to large objects and for migrating them between large objects. In our implementation, we chose a single large object for the whole index.

A developer of an access-method has no control over the locking of large objects, nor over logging and recovery. For example, it is not possible to unlock a large object storing some internal node while traversing a tree. If the repeatable-read isolation level is set, even the shared locks on large objects will be released only when a transaction commits. This implies that the concurrency control and recovery protocols of Kornacker et al. cannot be implemented using large objects.

Storing index data in a regular operating system file implies that all concurrency control and recovery protocols must be implemented by the access-method developer. It seems that it is possible to implement concurrency control integrated with the transaction management of Informix Server using transaction-end callbacks. However, there are no means to integrate the access-method recovery with the Informix Server's recovery subsystem. Although, using an operating system file, the developer has the freedom to implement any desirable concurrency control and recovery protocols, their implementation are complicated and time-consuming tasks. Neither the DataBlade API nor the Virtual Index Interface API provide any low-level services to assist.

Summarizing, sbspaces provide too high-level and too inflexible services for storing index data, rendering "industrial strength" concurrency control and recovery impossible. On the other hand, the external-file option is also not attractive because the APIs provide very little support for implementing concurrency control and recovery.

6. Implementation

In this section, we briefly describe the tools used and the main implementation tasks that were necessary developing the GR-tree DataBlade. More details about these subjects, as well as some hints about the coding guidelines and testing options are given in the full version of the paper [5].

6.1. Tools Used

Informix provides a set of tools, called the DataBlade Developer's Kit, that support DataBlade development. The core tool is BladeSmith, a GUI tool that allows a developer to define and manage different types of DataBlade objects, such as data types and routines [10]. Based on the definitions of these objects, BladeSmith automatically generates C source code (type-support functions and skeletons of all routines), SQL scripts, test files, and installation files for a DataBlade.

The DataBlade developer has to flesh out the provided skeletons and to compile the code, building a shared library (or a dynamic link library under Microsoft Windows) that will be loaded by the server when the DataBlade is used. The SQL scripts contain the code for registering and un-registering all DataBlade objects and their interrelationships in the system catalog tables. The scripts are run by BladeManager—a GUI tool for registering and un-registering DataBlade for databases in the Informix Server.

6.2. Tasks Performed

The tasks performed when implementing the GR-tree DataBlade are given in Table 4. They required a total of about 4.5 person-months of efforts, but the access-method core was implemented in C++ beforehand. The access method purpose functions turned into a layer connecting the already-implemented access-method core to the Informix Server.

All purpose functions were implemented except `am_scancost()`, `am_stats()`, and `am_check()`; these three are not necessary for an access method to be operational. Most of the implemented purpose functions required only little coding because their main tasks were to invoke already-implemented GR-tree-core functions. The areas that required more additional coding were the manipulation of BLOBs, the large object storing the index, and the manipulation of the qualification descriptor. The main design tasks for the writing of the purpose functions included considerations on how to re-use search information during the index scan (`am_getnext()` returns one qualifying row at a time) and how to implement deletions [5].

Being quite useful for addition of new data types, BladeSmith provides virtually no support for developing access method DataBlades. The purpose functions and operator-class functions are defined as regular routines in a BladeSmith's project. Thus, only their prototypes were generated in a C source file. Function bodies and the SQL code for registering and un-registering an access method and operator classes have to be hand-written by the developer.

7. Conclusions

Prompted by the need for the effective and efficient management of new kinds of complex data, the major DBMS vendors are proposing solutions that enable the users themselves to extend the DBMS. In the Informix Server, user-defined access methods can be created as DataBlades. We implemented the GR-tree index for now-relative bitemporal data as a DataBlade prototype with the goals of assessing the applicability of Informix for such a task; of identifying the weaknesses and strengths of the Informix Server architecture and its APIs and services supporting user-defined access methods; and of measuring the required development efforts.

Our experience shows that a prototype of an access method DataBlade can be developed relatively fast, provided that the index structure and algorithms are already in place. One of the obstacles that any access-method DataBlade developer currently faces is the lack of good documentation on the topic. As of this writing, the "Virtual-Table Interface Programmer's Manual" [12] is on hold, and the "Virtual-Index Interface Programmer's Manual" [11] is access restricted.

A major technical challenge developing a tree-based access method DataBlade is the implementation of efficient concurrency control and recovery mechanisms. Currently, Informix provides two possibilities for index storage: an index can be stored as a regular operating system file or as

Task	Complexity	LOC
Adapting the existing code to the DataBlade coding guidelines.	low	—
Defining the structure of the opaque type.	average	—
Including UC and NOW handling in opaque-type support functions.	low	30
Writing operations on the opaque type.	low	30
Designing the operator class framework.	high	—
Writing access method purpose functions.	high	1020
Writing BLOB manipulation functions.	average	280
Writing functions manipulating the qualification descriptor.	average	120

Table 4. Tasks, their Complexity, and Required Lines of Code

one or several so-called large objects inside Informix. While the first option is too "low-level," leaving implementation of all concurrency control and recovery mechanisms to the developer, the second is too "high-level," providing automatic locking for large objects that is unlikely to be efficient in a multi-user environment for access methods.

The operator class framework makes it possible to use several variations of the same access method, but it is not easy to understand, and may be improved. Different options concerning the usage of operator classes (cf. Sections 4 and 5.2) are not immediately clear and are not adequately discussed in the documentation. Strategy functions can take only single-column arguments; in our case, this restriction forced us to define the bitemporal extent of a tuple as one data type.

We feel that the existing framework for extending the Informix DBMS kernel with user-defined access methods represents a great advance, but it is still only a first step. Following the ideas of Hellerstein et al. [8] and Aoki [1], a generic, extendible tree-based access method with "industrial strength" concurrency control and recovery protocols could be integrated into the kernel of the DBMS. Such a generic access method would support a broad class of tree-based access methods by providing a simple, high-level extension interface that isolates the primitive operations required to construct new access methods [1]. It is also possible to implement a generic access method as a DataBlade and use specially designed operator classes to extend it.

Acknowledgements

This research was supported in part by the Danish Technical Research Council through grant 9700780, by the Chorochronos project, funded by the European Commission DG XII, contract no. FMRX-CT96-0056, and by a grant from the Nykredit Corporation.

References

[1] P. M. Aoki. Generalizing "Search" in Generalized Search Trees. *Proceedings of ICDE*, pp. 380–389 (1998).

[2] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Inf.*, 9:1–21 (1977).

[3] N. Beckmann et al. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of ACM SIGMOD*, pp. 322–331 (1990).

[4] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. *Proceedings of VLDB*, pp. 345–356 (1998).

[5] R. Bliujute, S. Saltenis, G. Slivinskas, and C. S. Jensen. Developing a DataBlade for a New Index. TIMECENTER Technical Report TR-29 (1998).

[6] J. Clifford et al. On the Semantics of "NOW" in Databases. *ACM TODS*, 22(2):171–214 (1997).

[7] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pp. 47–57 (1984).

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *Proceedings of VLDB*, pp. 562–573 (1995).

[9] INFORMIX-Universal Server DataBlade API User's Guide. (1997)

[10] INFORMIX. DataBlade Developers Kit User's Guide. (1997)

[11] INFORMIX. Virtual-Index Interface Programmer's Manual (1997).

[12] INFORMIX. Virtual-Table Interface Programmer's Manual (1997).

[13] C. S. Jensen and R. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4):311–352 (1996).

[14] C. S. Jensen et al. A Consensus Glossary of Temporal Database Concepts. In *Temporal Databases: Research and Practice*, O. Etzion, S. Jajodia, and S. Sripada (eds), Springer-Verlag, pp. 367–405, (1998).

[15] M. Kornacker and D. Banks. High-Concurrency Locking in R-Trees. *Proceedings of VLDB*, pp. 134–145 (1995).

[16] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *Proceedings of ACM SIGMOD*, pp. 62–72 (1997).

[17] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE TKDE*, 10(1):1–20 (1998).

[18] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM TODS*, 6(4):650–670 (1981).

[19] B. Salzberg and V. J. Tsotras. A Comparison of Access Methods for Temporal Data. TimeCenter TR-18 (1997). To appear in *ACM Computing Surveys*.

[20] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM TODS*, 12(2):247–298 (1987).

[21] R. T. Snodgrass et al. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers (1995).