Developing a DataBlade for a New Index

Rasa Bliujūtė, Simonas Šaltenis, Giedrius Slivinskas, and Christian S. Jensen

September 2, 1998

TR-29

A TimeCenter Technical Report

Title — Developing a DataBlade for a New Index

Author(s) — Rasa Bliujūtė, Simonas Šaltenis, Giedrius Slivinskas, and Christian S. Jensen

Publication History — September 1998. A TIMECENTER Technical Report.

Any software made available via TIMECENTER *is provided "as is" and without any express or implied warranties, including, without limitation, the implied warranty of merchantability and fitness for a particular purpose.*

Abstract

Many current and potential applications of database technology, e.g., geographical, medical, spatial, and multimedia applications, require efficient support for the management of data with new, complex data types. As a result, the major DBMS vendors are stepping beyond the support for 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 what is perhaps currently the 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 research community, because these techniques can be integrated into DBMSs more easily. This facilitates dissemination of research results as well as the transition from research results to products.

The paper describes a prototype implementation of a new access method, termed the GR-tree [BJSS98], as an Informix DataBlade. Based on the R*-tree [BEC90][1], this tree indexes now-relative bitemporal data, which is data with associated valid-time and transaction-time values [SA85] [JS96]. 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. Valid time is meaningful and necessary for a wide range of applications, and transaction time is particularly useful in applications where trace-ability or accountability are important. 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. Many real-world databases contain a significant portion of this type of data.

The paper describes 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

[1] The R*-tree is an improved version of the R-tree originally proposed by Guttman [GUT84].

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, the specific coding tasks, and debugging and testing.

The presentation is structured as follows. Section 2 explains what bitemporal data is and how it may be represented, upon which Section 3 proceeds to briefly describe 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. Valid time captures when a tuple's information is true in the modeled reality, and transaction time captures when a tuple is current in the database [SA85, JS96]. 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.

TQuel's four-timestamp format [SNO87] (4TS) is the most popular format for bitemporal data representation. With this format, each tuple has a number of non-temporal attributes and four time attributes: `VTbegin` and `VTend`—the times when the tuple's information became and ceased to be true in the modeled reality; `TTbegin` and `TTend`—the times when the tuple became and ceased to be current in the database.

A tuple is now-relative if its information is valid until the current time or if the tuple is part of the current database state. This is represented in the 4TS format by the use of variables, which denote the current time, for the time attributes `VTend` and `TTend` [CLI97]. The variable UC (denoting "until changed") is used for `TTend`, and the variable NOW is used for `VTend`. Table 1 exemplifies the 4TS format. The time granularity is a month, and the current time (CT) is assumed to be 9/97.

	Employee	Department	TTbegin	TTend	VTbegin	VTend
(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

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

Specific constraints apply to insertions, deletions, and modifications of tuples in a bitemporal database. When inserting a new tuple, the constraints `VTbegin` $\leq$ `VTend` and `VTbegin` $\leq$ 'current time' if `VTend` is equal to NOW apply to valid time; and the constraints `TTbegin` $=$ 'current time' and `TTend` $=$ UC apply to transaction time. Any *current* database tuple can be deleted or modified. Deleting a tuple, the `TTend` value UC is changed to the fixed value 'current time'-1[2], logically deleting the tuple (e.g., Tuple (2)). Tuples are not physically deleted. A modification is modeled as a deletion followed by an insertion (e.g., an update led to Tuple 4 and Tuple 5).

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 [JS96]. Cases 1–5 in Figure 1 illustrate *bitemporal regions* of

[2] We use closed intervals and let [`TTbegin`, `TTend`] denote the interval that includes `TTbegin` and `TTend`.

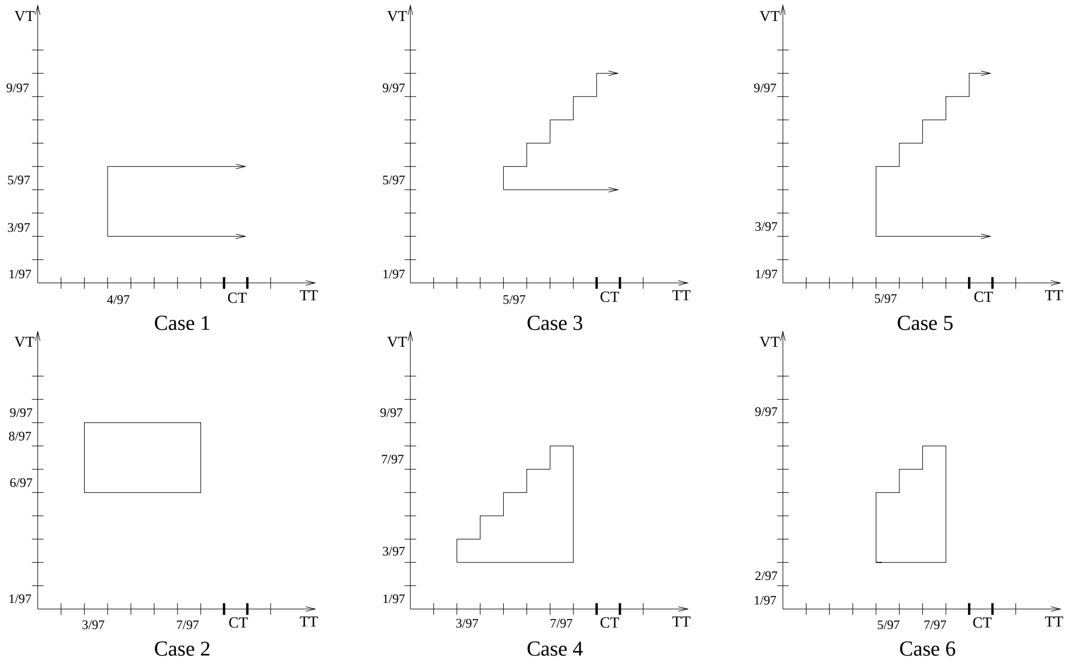

Figure 1: Bitemporal Regions

Tuples (1–4) and (6), respectively.

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. In this case, the valid-time end must be a ground value (Tuple (2), Case 2); otherwise, the valid-time end, which would extend to the current time, would initially be smaller than the valid-time start, violating the second insertion constraint of the valid time. If, at some time, a tuple is logically deleted, the bitemporal region stops growing (Tuples (2), (4); Cases 2, 4, 6).

Stated generally, we obtain six combinations of time attributes for which the bitemporal regions are qualitatively different (Figure 1), as illustrated in Figure 2 where 'tt1', 'tt2', 'vt1', and 'vt2' denote ground values that satisfy the constraints given above.

	`TTbegin`	`TTend`	`VTbegin`	`VTend`	
Case 1	tt1	UC	vt1	vt2	
Case 2	tt1	tt2	vt1	vt2	
Case 3	tt1	UC	vt1	NOW	(tt1=vt1)
Case 4	tt1	tt2	vt1	NOW	(tt1=vt1)
Case 5	tt1	UC	vt1	NOW	(tt1>vt1)
Case 6	tt1	tt2	vt1	NOW	(tt1>vt1)

Figure 2: Possible Combinations of Time Attributes

Thus, the time associated with bitemporal data can be represented as two-dimensional regions and can be indexed using adapted spatial indices. For a description of the proposed indices for bitemporal data, including their performance and other properties, the reader is referred to [BJSS98]. An essential challenge in indexing bitemporal data is to properly handle now-relative intervals. The next section briefly presents the GR-tree index which contends well with this requirement and which 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, which we first describe briefly. The R*-tree is a spatial index consisting of nodes organised 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 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 3 shows minimum bounding rectangles of spatial objects (to the left) and the corresponding tree (to the right).

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.

Dead space, the space in the minimum bounding rectangle not occupied by any enclosed rectangle, and the *overlap* between the minimum bounding rectangles of the nodes at each level of the tree are two important properties that define the “goodness” of a tree (see Figure 3). Minimizing overlap reduces the I/O-incurring branching of search into several subtrees. Minimizing dead space reduces the probability that queries unnecessarily access disk pages, eventually finding no qualifying data. Consider the query rectangle given in Figure 3: it overlaps with minimum bounding rectangles R1 and R2; thus, to find the qualifying entries we need to read two nodes that are bounded by these rectangles. However, the query rectangle does not overlap with any of the rectangles enclosed in R1.

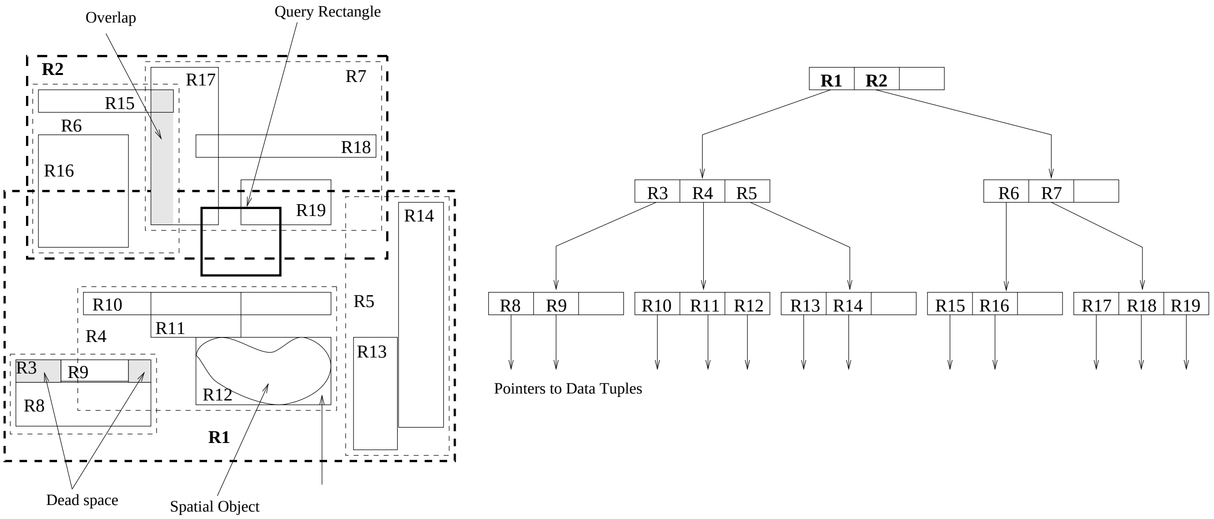

Figure 3: Example of the R*-tree

Because the R*-tree cannot handle the growing bitemporal regions presented in Section 2, it was modified, leading to the GR-tree [BJSS98]. 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. Those minimum bounding regions can be either rectangles or stair-shapes. Minimum bounding regions grow when the regions inside them grow.

Figure 4 illustrates the bounding of a node with a rectangle (a) and with a stair-shape (b). In case (b), it is reasonable that node 2 is bounded with a stair-shape because none of the included regions extend above the line $y = x$.

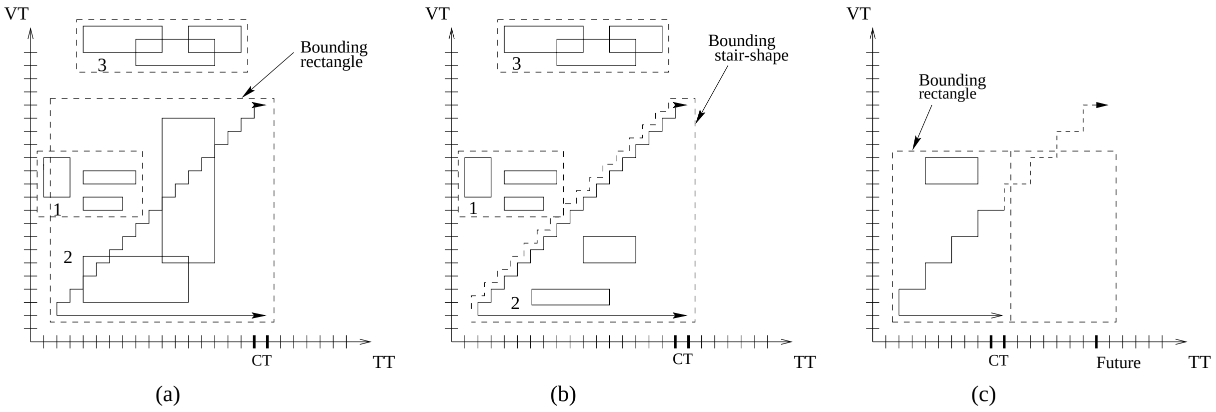

Figure 4: Graphical Representation of (a) a Minimum Bounding Rectangle of a Node, (b) a Minimum Bounding Stair-Shape of a Node, and (c) a ”Hidden” Growing Stair-Shape

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. The possible combinations of the four timestamps are shown in Figure 2, and they encode the bitemporal regions in Figure 1.

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. Note that timestamps (tt1, UC, vt1, NOW) represent a stair-shape in a leaf-node entry, but can represent a stair-shape *or* a rectangle growing in both transaction and valid time in an entry of a non-leaf node. For instance, if a leaf node contains a growing stair-shape and a rectangle that extends above the $y = x$ axis, the node has to be bounded with a rectangle growing in both dimensions (see Figure 4(a)). The “`Rectangle`” flag is needed to distinguish between these two possibilities. A sample tree is given in Figure 5. Node 2 contains a growing stair-shape, and its bounding region (in the second entry of the root) is also a growing stair-shape.

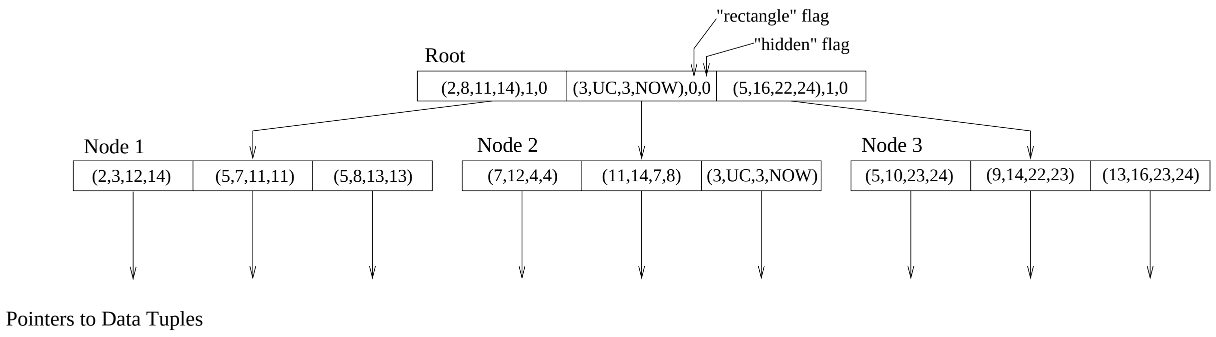

Figure 5: GR-tree Structure

A small growing stair-shape may be placed together with other regions in a larger bounding rectangle having a fixed valid-time end (that is bigger than the current time). One day, the stair-shape will outgrow its bounding rectangle, making this rectangle invalid, see Figure 4(c). The flag “`Hidden`” is used to track such stair-shapes.

The GR-tree is better than the other proposed indices because its structure leads to smaller overlap and dead space. 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. Since the flag “`Hidden`” in an entry carries information about the region encoded by this entry, the following algorithm must be used to adjust the `VTend` value of the region whenever it is involved in the computations.

```
IF flag Hidden is set AND VTend is fixed AND VTend is less than the current time
THEN set VTend to NOW
```

The shape of the region encoded by a non-leaf node entry can be determined from the "`Rectangle`" flag or from the variable NOW, if the entry is in a leaf node. The variable UC indicates whether the region is static or growing. The GR-tree algorithms utilize this information, but the actual values of UC and NOW, i.e., the top-right corner of a region, are also needed. These values depend on the current time and can be found as follows.

```
IF TTend is equal to UC
THEN set TTend to the current time
IF VTend is equal to NOW
THEN set VTend to TTend
```

New insertion algorithms that take into account spatial and temporal differences between bitemporal regions are designed for the GR-tree. In addition, a time parameter, capturing the development over time of entries, is introduced in these algorithms. Reference [BJSS98] contains an in-depth description of the GR-tree algorithms.

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.[3] 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 [DBAPI97], the Virtual-Table Interface API [VTI97], and the Virtual-Index Interface API [VII97] 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 ("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, have to 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 a data type for time extents in the GR-tree DataBlade is given in Section 5.1, and the implementation of this type is covered in Section 6.3.

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 have to be written, compiled, and registered. These functions are to be coded in C/C++ 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.

[3] The abbreviation "Informix Server" will be used throughout.

```
CREATE FUNCTION grt_open(pointer)
RETURNING int
EXTERNAL NAME ''usr/functions/grtree.bld(grt_open)''
LANGUAGE c;
```

The tasks that the purpose functions are responsible for and the names of the corresponding purpose functions are given in Table 2. Only the `am_getnext()` function is mandatory.

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

If the Informix Server determines that a table specified in an SQL statement should be accessed via a virtual index, it which are dynamically loads and executes the appropriate purpose functions. Figure 6 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.

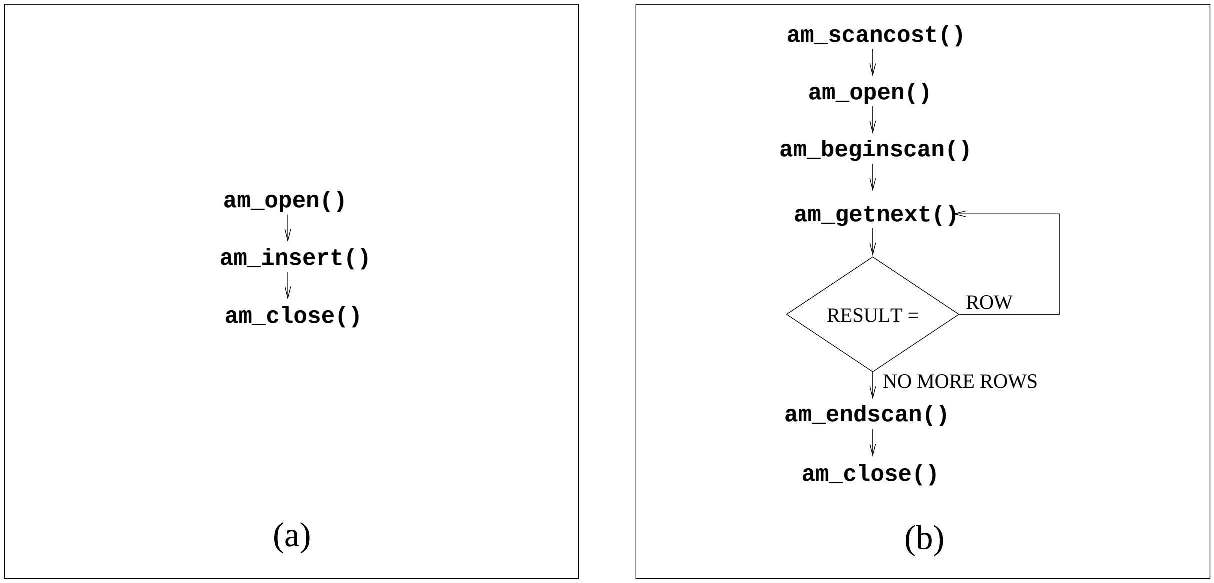

Figure 6: Access Method Purpose Functions Called for (a) `INSERT` and (b) `SELECT` Statements

A number of structures, termed *descriptors*, are used in the purpose functions. These 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 can be 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 Step 5).

```
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 an index structure, but are not data-type specific. 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 7), 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.

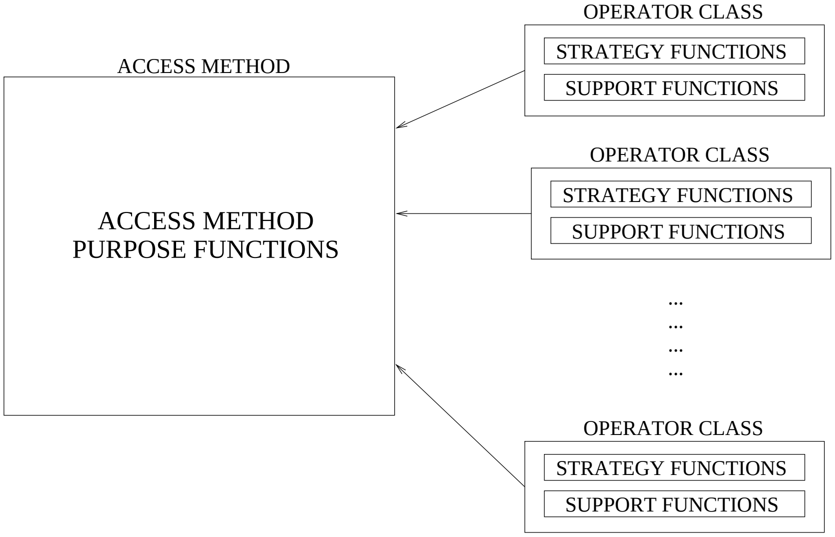

Figure 7: Association Between an Access Method and Operator Classes

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 6(b). Function `am_getnext()` dynamically resolves which strategy function is used and invokes that function on index entries to find the qualifying regions.

To add support for a new data type to an existing access method, new additional strategy functions with the same names, but new argument types, should be written and registered. This way, in Informix terms, 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 a 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 leave 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 a simple 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);
```

The operator class can be registered as a default operator class of the access method.

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. If an operator class is not specified for a column in the `CREATE INDEX` statement, a default operator class of an access method will be used. 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;
```

The CREATE SECONDARY ACCESS_METHOD statement enters access method information into the system catalog table SYSAMS. The CREATE INDEX statement adds index information to the system catalog tables SYSINDICES and SYSFRAGMENTS.

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. Using the GR-tree DataBlade implementation as an example, this section reveals some 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. Section 5.4 tells about the usage of the current time value. Section 5.5 reveals some aspects that are important when considering deletions.

5.1 Physical Representation of a Time Extent

The GR-tree indexes the time extents associated with the database records. These extents are formed by four timestamps: TTbegin, TTend, VTbegin, and VTend. We have so far assumed that each timestamp 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 TTbegin, TTend, VTbegin, and VTend—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 TTend and VTend, 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 UC and NOW values, 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 8. 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 VTend is NOW and Julie’s bitemporal extent is a stair-shape, which does not overlap with the given query region.

Thus, any “bitemporal” function f(timeextent1, timeextent2) (e.g., “f” could be “overlaps”) cannot be replaced by two functions f1(valid_interval1, valid_interval2) and f2(transaction_interval1, transaction_interval2).

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)

Name	Department	TTbegin	TTend	VTbegin	VTend
Julie	Sales	3/97	7/97	3/97	NOW

Table 3: The EmpDep Relation

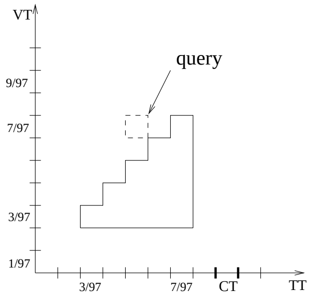

Figure 8: Time Extent of the Julie Record in the EmpDep Relation

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*. This structure is restricted to accommodate only single-column predicates:

- f(column, constant)
- f(constant, column)
- f(column)

This 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 `DATE` type. Many other possibilities exist, one being the 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 index defined on it. If such an index exists 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 index 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 a sample query.

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

Function Overlaps() works on two GRT_TimeExtent_t objects and its prototype is given below:

```
mi_boolean Overlaps(GRT_TimeExtent_t* ext1, GRT_TimeExtent_t* ext2);
```

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. However, 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 representing time extent "12/10/95, UC, 12/10/95, NOW".

Note that the four timestamps in an internal-node entry do not uniquely identify 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 an 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 with 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 based on the values of these arguments. 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, an *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 during insertions. The upper levels of the subtree are locked so that only readers can still access them [BS77]. To achieve much higher levels of concurrency, B-link trees [LY81] and R-link trees [KB95] were proposed, and Kornacker et al. [KMH97] 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-described 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 the 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 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 is a complicated and time-consuming task. 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.

5.4 Current Time and Transactions

The algorithms of the GR-tree use the current-time value for resolving the UC and NOW variables stored in index entries.

The simplest solution is to use a constant current-time value during a single statement for which the index has to be used. This implies getting this time value when the index is opened (in the `am_open` purpose function) and storing it in the structure associated with the open index (the so-called *index descriptor*).

If the constant current-time value is wanted not only for a single statement, but for the entire duration of a transaction, the only possible moment to get it is the first time the index is used during the transaction (the first time the `am_open` purpose function is called). It is not possible to use, for example, the begin time of the transaction, because the DataBlade API does not provide means of capturing a transaction-begin event in a DataBlade. The obtained current-time value can be stored in the named memory allocated from a server and identified by the session id, under which the transaction is running. A transaction-end callback should be registered to free the allocated memory. The GR-tree DataBlade uses this approach.

5.5 Deletions

The deletion procedure works as follows: entries satisfying a qualification predicate are retrieved and deleted one by one. The `am_getnext()` function is the one retrieving qualifying entries. The natural wish is to reuse the work of a previous search, e.g., if a node that contains several qualifying entries is found, and the first qualifying entry is deleted, the next time entering an `am_getnext()`, we do not want to traverse the tree again to find the same node. However, if a node from which an entry is deleted gets underfull, the tree has to be condensed, i.e., the remaining entries have to be re-inserted. This re-insertion changes the structure of the tree, and the tree-traversal information from the previous search can not be reused.

So deletions become time-consuming because every time after we find a qualifying entry, the tree has to be traversed from the root to find the next qualifying entry. One possible optimization is to postpone re-insertions and save entries that should be re-inserted in main memory. But if many entries fulfill the deletion qualification, many nodes may get underfull, and a big amount of entries will have to be re-inserted. Chances then are that they will not all fit in main memory. For our prototype, we chose a compromise in-between the two: we decided to restart scanning of the index only when the tree is actually condensed.

In multi-user environments, deletions in an index get more complicated due to the presence of concurrency control. To achieve the repeatable-read isolation level, entries are not physically deleted, but only marked for deletion (the actual deletion is performed only at transaction commit).

Re-insertions caused by deletions lower the availability of the index. This problem may be addressed by not performing re-insertions at all or by allowing nodes with only few entries. But it is also not desirable to have many poorly filled nodes because this negatively affects search performance. Deletions of nodes also become complicated because concurrent operations on an index may still have pointers to nodes to be deleted. Detailed discussion of these issues is provided elsewhere [KMH97].

Sometimes vacuuming will have to be performed to delete all data that is more than, for example, five years old. Possibly, a big amount of entries then will have to be deleted, and it clearly will be inefficient to use the deletion procedure described above. A straightforward solution is to drop the index and then create it from scratch using a bulk loading algorithm. Alternatively, a bulk deletion algorithm may be provided.

6 Implementation

In this section, we describe the main implementation tasks that were necessary developing the GR-tree DataBlade. As a background for this, in Section 6.1, we present the tools used, and Section 6.2 covers the coding guidelines for Informix DataBlades. The tasks performed are sketched in Section 6.3, and Section 6.4 gives hints about testing and debugging options.

6.1 Tools Used

Informix provides a set of tools, called DBDK (DataBlade Developer's Kit), that support DataBlade development. The core component of DBDK is *BladeSmith*, a GUI tool that allows a developer to define and manage different types of DataBlade objects, such as data types, type casts, routines, interfaces between DataBlades, errors, SQL files, and client files [DBDK97]. Based on the definitions of these objects, BladeSmith automatically generates C source code, SQL scripts, test files, and installation files for a DataBlade. A separate BladeSmith project is created for each DataBlade that is being developed.

The source code generated by BladeSmith consists of one header file, one C source file, and a make file. BladeSmith generates the skeletons of all routines defined in a BladeSmith project and the skeletons of the routines necessary to implement newly defined data types and type casts. 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 whenever the DataBlade is used. The process is iterative, i.e., a developer can modify defined objects, and BladeSmith will regenerate the code, merging it with the code added by the developer.

The BladeSmith's automatic code generation is very useful for routine tasks, but during the development of our prototype DataBlade, we found some inconveniences related to it. BladeSmith generates source code as one source file. Usually, a software development project involves far more than one source file. For example, it would be natural to divide the source code of access method purpose functions and the source code of specific operator-class functions into different source files. In addition, the header file generated by BladeSmith cannot be changed, because changes are lost each time the source code is regenerated. To avoid the above-mentioned inconveniences, we chose to avoid the iterative development and to generate the source files only once, when the definition of data types and routines was complete.

BladeSmith also generates the SQL scripts that are run to register and un-register a DataBlade for a specific database in the Informix Server. The scripts contain the automatically generated code for registering and un-registering all DataBlade objects and for recording their interrelationships in the system catalog tables; a developer can provide additional code. The scripts are run by DBDK BladeManager—a GUI tool for registering and un-registering DataBlade for databases in the Informix Server. We found this framework to be very convenient when a DataBlade is being developed, because during testing it has to be registered and un-registered multiple times.

6.2 Coding Specifics

DataBlade modules execute in a so-called virtual processor in Informix Server. Special coding guidelines must be followed to ensure that a DataBlade module executes safely, efficiently, and without disturbing other processes of Informix Server.

Because of the multi-threaded Informix Server environment, DataBlade code should be thread-safe. Global and static variables should not be used. If some information needs to be preserved between function calls, server shared memory should be allocated and used. One should also avoid any library or operating system calls that could block or any library calls that use allocated memory between calls. A DataBlade module should not raise, handle, or mask operating system signals. Because of the non-preemptive nature of Informix Server, DataBlade functions that require a lot of CPU time should regularly yield processor (with *mi_yield* calls).

To improve the portability and extensibility of DataBlade code, DataBlade API data types should be used in variable definitions. For instance, data type `mi_integer` should be used instead of `int`.

Memory can be allocated and deallocated only using special DataBlade API functions that allocate memory in the space shared by all server processes. The allocated memory can have different durations, e.g., `PER_FUNCTION` or `PER_STATEMENT`. The Informix Server automatically frees allocated memory when the specified duration is exceeded.

6.3 Tasks Performed

The tasks performed when implementing the GR-tree DataBlade are given in Table 4. The access-method core was implemented in C++ beforehand, and therefore the tasks included learning the neccessary APIs, adapting the existing code according to the DataBlade coding guidelines, creating an opaque data type for time extents, writing access method purpose functions (which turned into a layer connecting the already-implemented access-method core to the Informix Server), and writing special functions to manipulate BLOBs and qualification descriptors. These implementation tasks required a total of about 4.5 person-months of efforts.

Adapting the existing code according to the DataBlade coding standards was not very difficult. Object-oriented programing naturally leads to code with no global or static variables. *New* and *delete* operators, used when programming in C++, had to be properly redefined to call DataBlade API memory management functions.

When creating the new opaque data type `GRT_Time_Extent_t`, most time was spent considering the data-type design options that were discussed in Section 5.1. After that, the structure of the type was specified, and so-called *type support functions* that work with this structure were implemented. These included:

1. Text input/output functions used to transform type values between their textual representation and the internal structure. The textual representation is used in SQL statements and in the results of SQL statements.

2. Binary send/receive functions used to transform type values between the internal structure and the representation used in the communication between a client and a server.

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

3. Text-file import/export functions making it possible to use the command `LOAD` for loading values of a new type from a text file to a table.

BladeSmith turned out to be particularly useful for defining the structure of the opaque type and for generating its support functions. According to the definition of the internal structure of the opaque type, BladeSmith automatically generated a type definition in a C header file and the code for the desired type support functions. However, it resulted in unnecessary repetition of code because, e.g., text-file import/export functions and text input/output functions perform very similar tasks. Such repetition is especially unwelcome when modifications of generated code are needed. We added to the generated code the handling of UC and NOW and the checking of the specific constraints that apply to values of the type (cf. Section 2).

The next step was to decide which operations to implement on the data type. We implemented only `Overlaps()`, but `Equal()`, `Contains()`, and `ContainedIn()` could be easily added. Then we had to decide what operator class framework should be used, i.e., whether functions operating with internal-node regions should be "hard coded" or not (cf. Section 5.2). We chose to declare only strategy functions operating on `GRT_Time_Extent_t` objects in the operator class and to "hard-code" all functions operating on internal-node regions, disabling extensions of the existing operator class and the creation of new ones.

The following purpose functions were implemented: `grt_create()`, `grt_drop()`, `grt_open()`, `grt_close()`, `grt_beginscan()`, `grt_getnext()`, `grt_rescan()`, `grt_endscan()`, `grt_insert()`, `grt_delete()`, and `grt_update()`. Because an access method is operational without the functions `am_scancost()`, `am_stats()`, and `am_check()`, these were omitted. Most of the implemented purpose functions required only little of 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, which is included as an argument in the scan descriptor passed to the `am_beginscan()`, `am_getnext()`, `am_rescan()`, and `am_endscan()` functions, and which contains the qualification, i.e., the part of the WHERE clause that is relevant to the index.

BLOB manipulation functions, including `Create()`, `Drop()`, `Open()`, `Close()`, `Read()`, and `Write()`, were written according to "INFORMIX-Universal Server DataBlade API User's Guide" [DBAPI97]. For the manipulation of the qualification descriptor, we had to code the logic for how to break a complex qualification (containing several strategy functions separated by `AND`'s or `OR`'s) into simple ones (containing each one strategy function) and for how to invoke appropriate strategy functions.

The main design tasks for the writing of the purpose functions included considerations on how to re-use search information during the index scan (`grt_getnext()` returns one qualifying row at a time) and how to implement deletions (cf. Section 5.5). Appendix A describes the purpose functions in more detail.

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.

6.4 Debugging and Testing

During debugging and testing of a DataBlade module, it may be necessary to restart a server; thus, it is recommended that debugging and testing is performed on a separate server. In addition, a so-called extended virtual processor class should be used during testing. Although sacrificing execution speed, this removes some coding restrictions (cf. Section 6.2) and allows the DataBlade module to be less well-behaved.

Our findings are that the extensive usage of trace messages is a good instrument for debugging a DataBlade module. Trace messages are directed to a special trace file and can be switched on or off selectively using trace classes and trace levels [DBAPI97].

It is also recommended to debug and test as much code as possible outside of the Informix Server, and integrate it into a DataBlade module only when it is correct.

7 Conclusions

Due to the need for efficient handling of new kinds of complex data, the major DBMS vendors are proposing solutions that allow the users themselves to extend a DBMS. In addition to an ability to have new data types, often efficient querying capabilities are required. User-defined access methods can be created as DataBlades in Informix Server. 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 task; of finding out the weaknesses and strengths of the Informix Server architecture, 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 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, ”Virtual-Table Interface Programmer's Manual” [VTI97] is on hold, and ”Virtual-Index Interface Programmer's Manual” [VII97] is also 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 one or several large objects in an sbspace. While the first option is "too low-level," leaving implementation of all concurrency control and recovery mechanisms to a developer, the second is "too high-level," providing an automatic locking for large objects, which may not be efficient in a multi-user environment.

The operator class framework makes it possible to use several variations of the same access method, but it is not an easily understandable and conceptually clean framework. 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, and, 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 is only the first step. Following the ideas of Hellerstein et al. [HNP95] and Aoki [AOK98], 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 the 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 [AOK98]. It is also possible to implement such a generic access method as a DataBlade and use specially designed operator classes to extend it.

References

[AOK98] P. M. Aoki. Generalizing "Search" in Generalized Search Trees. *Proceedings of ICDE*, pp. 380–389 (1998).

[BEC90] 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).

[BJSS98] R. Bliujute, C. S. Jensen, S. Saltenis, and G. Slivinskas. R-tree Based Indexing of Now-Relative Bitemporal Data. To appear in *Proceedings of VLDB* (1998)

[BS77] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Inf.*, 9:1–21 (1977).

[CLI97] J. Clifford et al. On the Semantics of “NOW” in Databases. *ACM TODS*, 22(2):171–214 (1997).

[DBAPI97] INFORMIX-Universal Server DataBlade API User’s Guide. (1997)

[DBDK97] INFORMIX. DataBlade Developer’s Kit User’s Guide. (1997)

[GUT84] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of ACM SIGMOD*, pp. 47–57 (1984).

[HNP95] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. *Proceedings of VLDB*, pp. 562–573 (1995).

[JEN98] 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).

[JS96] C. S. Jensen and R. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4):311–352 (1996).

[KB95] M. Kornacker and D. Banks. High-Concurrency Locking in R-Trees. *Proceedings of VLDB*, pp. 134–145 (1995).

[KMH97] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *Proceedings of ACM SIGMOD*, pp. 62–72 (1997).

[KTF98] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE TKDE*, 10(1):1–20 (1998).

[LY81] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM TODS*, 6(4):650–670 (1981).

[SA85] R. T. Snodgrass and I. Ahn. A Taxonomy of Time in Databases. *Proceedings of ACM SIGMOD*, pp. 236–246 (1985).

[SAM90] H. Samet. The Design and Analysis of Spatial Data Structures. *Addison-Wesley* (1990).

[SNO86] R. T. Snodgrass. Temporal Databases. *IEEE Computer*, 19(9):35–42 (1986).

[SNO87] R. T. Snodgrass. The Temporal Query Language TQuel. *ACM TODS*, 12(2):247–298 (1987).

[SNO95] R. T. Snodgrass et al. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers (1995).

[SRF97] T. Sellis, N. Roussopoulos, and C. Faloutsos. Multidimensional Access Methods: Trees Have Grown Everywhere. *Procedings of VLDB*, pp. 13–14 (1997).

[ST97] B. Salzberg and V. J. Tsotras. A Comparison of Access Methods for Temporal Data. TimeCenter TR-18 (1997). To appear in *ACM Computing Surveys*.

[VII97] INFORMIX. Virtual-Index Interface Programmer’s Manual (1997).

[VTI97] INFORMIX. Virtual-Table Interface Programmer’s Manual (1997).

A Access Method Purpose Functions

We give an overview of the GR-tree access method purpose functions in Table 5. Tree denotes an object implementing the GR-tree and having methods such as insert(), delete(), and search(). Method search() creates an object Cursor which stores a query predicate (qualification descriptor) and a tree-traversal information. Qualifying entries are retrieved by calling Cursor's method next()

Table 5: Access Method Purpose Functions

Function	**Steps Performed**
grt_create() *input:* a pointer to index descriptor td.	(1) Create object Tree and save its pointer in td. (2) If the grtree_am access method cannot handle types of columns specified in the CREATE INDEX statement, produce an error message and exit. (3) If the operator class specified in the CREATE INDEX statement cannot be used for the grtree_am access method, produce an error message and exit. (4) If there already exists an index using the grtree_am access method on the same columns and with the same user-defined parameters, produce an error message and exit. (5) Create a BLOB where the index will be stored. (6) Insert a new record containing index id, its fragment id, and the BLOB handle in the table associated with the grtree_am access method. (7) Open the BLOB.
grt_drop() *input:* a pointer to index descriptor td.	(1) Get a pointer to Tree object from td. (2) Drop the BLOB. (3) Delete Tree object. (4) Delete the corresponding record from the table associated with the the grtree_am access method.
grt_open() *input:* a pointer to index descriptor td.	(1) If the function was invoked right after grt_create(), exit. (2) Create object Tree and save its pointer in td. (3) Get the BLOB handle for the index from the table associated with the grtree_am access method. (4) Open the BLOB.
grt_close() *input:* a pointer to index descriptor td.	(1) Get a pointer to Tree object from td. (2) Close the BLOB. (3) Delete Tree object.

continued on next page

continued from previous page

Function	**Steps Performed**
`grt_beginscan()` *input:* a pointer to scan descriptor `sd`.	(1) Get qualification descriptor `qd` from `sd`. (2) Get index descriptor `td` from `sd`. (3) Create `Cursor` object (by calling `Tree`'s `search()` method) to store query predicate (`qd`) and tree-traversal information. (4) Save a pointer to `Cursor` in `td`.
`grt_rescan()` *input:* a pointer to scan descriptor `sd`.	(1) Get index descriptor `td` from `sd`. (2) Get a pointer to `Cursor` object from `td`. (3) Reset `Cursor`.
`grt_getnext()` *input:* a pointer to scan descriptor `sd`. *output:* a pointer to the id of the row to be returned `retrowid`, a pointer to the row (consisting of indexed fields) to be returned `retrow`.	(1) Get index descriptor `td` from `sd`. (2) Get a pointer to `Cursor` object from `td`. (3) Get a qualifying entry by calling `Cursor`'s `next()` method. (4) Get `rowid` and `fragid` from the retrieved entry and use them to form `retrowid`. (5) Get row descriptor `rd` from `td`. (6) Form `retrow` from the retrieved entry using `rd`.
`grt_endscan()` *input:* a pointer to scan descriptor `sd`.	(1) Get index descriptor `td` from `sd`. (2) Get a pointer to `Cursor` object from `td`. (3) Delete `Cursor`.
`grt_insert()` *input:* a pointer to index descriptor `td`, a pointer to the row to be inserted `newrow`, a pointer to the id of that row `newrowid`.	(1) Get a pointer to `Tree` object from `td`. (2) Form the entry to be inserted from the `newrow` and the `newrowid`. (3) Insert the entry by calling `Tree`'s `insert()` method.
`grt_delete()` *input:* a pointer to index descriptor `td`, a pointer to the row to be deleted `oldrow`, a pointer to the id of that row `oldrowid`.	(1) Get a pointer to `Tree` object from `td`. (2) Get a pointer to `Cursor` object from `td`. (3) If there is no `Cursor` object, create `Cursor` object (by calling `Tree`'s `search()` method) and, using it, find the entry that contains `oldrowid`. (4) Delete the entry by calling `Tree`'s `delete()` method. (5) If tree is condensed, reset `Cursor`. (6) If `Cursor` was created in step (3), delete `Cursor`.
`grt_update()` *input:* a pointer to index descriptor `td`, a pointer to the row to be deleted `oldrow`, a pointer to the id of that row `oldrowid`, a pointer to the row to be inserted `newrow`, a pointer to the id of that row `newrowid`.	(1) Invoke `grt_delete()` with `td`, `oldrow`, and `oldrowid` as arguments. (2) Invoke `grt_insert()` with `td`, `newrow`, and `newrowid` as arguments.