Extensible Motor of a Object-Relational DBMS: Design and Implementation

Shi-guang Ju, Sergio V. Chapa and Wei-he Chen
Computer Dept., Jiangsu University of science & technology, China
Computer Dept., CINVESTAV, IPN, Mexico
Jushig@delta.cs.cinvestav.mx
Schapa@alpha.cs.cinvestav.mx

Abstract

A Object-Relation DBMS design and implementation is introduced. It is a hybrid of relational database management and object-oriented technique. It will support the ORSQL that is the super-set of SQL and is applied to those application involved of complex datatype and complex query. In the paper, we emphatically describe the system architecture and its mechanism of storage, access and query that appropriately apply to the complexity, dynamic and high-dimensions datatype.

1 Introduction

As the social demand for database use expands, new application fields of database are emerging continuously. They have imposed many new demands on data manager.

- To store and manage all kinds of complex objects, which have complex structure in common, i.e., some of attributes, describing an object, may be another object.
- To add new datatypes and specific operations in some applications. So it is naturally for a DBMS to support user-defined function and user-defined datatype.
- To support dynamic schema modification. For example, in an engineering design system, the design data are increased and modified step by step. It is no surprising that some entity's structures and relationships between themselves are not considered in their previous design stages. In the case, we have to modify the previous schema. And so DBMS must have the capability of dynamic schema modification.
- To process multiple dimensions data.

0-7695-0393-4/99 $10.00 © 1999 IEEE

Of these new features, both traditional relation DBMS and object-oriented DBMS are difficult to manage these data efficiently. It is commonly recognized that this model is inadequate for future data processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". However, as mentioned, this simplicity often makes the implementation of certain applications very difficult. So we have to study and develop some new type DBMS systems.

One of the most important leading directions of new advancement in DBMS is combining relational technique with object-oriented technique to form a new type database management system.

Definition 1. A system has the capabilities of relational DBMS and support object-oriented, mainly extending classes, inheritance, types and functions. And providing the powers: constraints, triggers, rules, transaction integrity and large objects. It is called Object-Relation Management Database System(ORDBMS).

A ORDBMS system that is designed and implemented, as definition 1, as a hybrid of relational database management and object-oriented technique. It will support the ORSQL that is the super-set of SQL and is applied to those application involved of complex datatype and complex query.

In this paper it is emphatically described that the object-relational DBMS system developed by ourselves from the following aspects: storage mechanism and access mechanism. And its query mechanism will be introduced in another paper.

2 Architecture of ORDBMS

We construct our ORDBMS in a two-layers architecture as shown in Figure 1. The user-oriented layer mainly deals with the translation and process of languages. Those are: Object-Relational query language (ORSQL), Object-Relation data manipulation language (ORDML) and Object-Relational data definition language (ORDDL). By parsing all kinds of users command statements, views transformation and query optimization must be done. Then the executable codes will be generated.

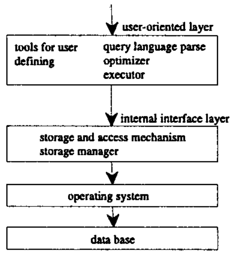

Figure 1: Two layers architecture of ORDBMS

The internal interface layer is running on the storage manager. And the storage manager must be a kernel that supports both tables storage and objects storage.

It realizes the transform from logic storage to physical storage by relying on the page management, buffer management and memory management of its operating system. The tasks of storage and access mechanism are to transform the upper layer's operations set to single object operation or single record operation.

3 Storage and access mechanism

3.1 Data Structure

The storage and access manager's core is needed to have an efficient storage and access data structure. It have to store relational information and manage efficiently all kinds of objects information. For this, we proposed a new data structure which can logically be divided into two layers as shown in Figure 2.

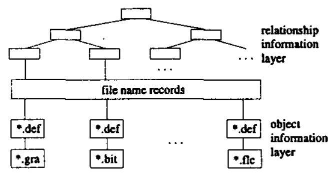

Figure 2: Two-layers data structure

The first layer is relational information layer. It is used mainly to store relational information between object classes. Obviously, the information is in text and can be represented by table structure and using B-trees index to carry out the storage and access operations. The second is object data layer in which all kinds of heterogeneity data, such as graphics, image, video, audio and the other object, are being stored. In order to identify the datatype of the object's attributes, we adopt the postfix of a file name to represent the object class. Such as ".gra" is used to representing graphics class and ".wav" is used to representing sound class.

In the relational information layer, the content of leaf node are some file names. In the essentially, these file names will be used as pointers which are pointing to the correspondent describe structure files (the postfix of file name is ".def") and related object instance data files (the postfix of file name is ".gra" and so on). By this means, the two layers information is associated together.

3.2 Storage of class and objects

Class is only existing as the description of a group of objects. In our system, class is stored as a specific object. We store class and object instances separately. On the other hand, one class always point out its object instances' position. All classes are being grouped as index structure. Their roots are kept in the database system directory table. So when accessing an object and sending messages to an object, we must find the correspondent description of class according to the root of class index in the system directory table to do semantic interoperation and methods call. The persistent storage format of class is shown as Table 1.

Flag	Name	CID	Ref. times	Index 1, 2	Instance index	Super-class table

Table 1: Storage format of a class

The CID field is used to storing identification of the class. If a class is used as an attribute of another class or a super class, we call this class to be referenced one time. The "Ref. times", in Table 1, just is used for recording the times of the class being refereed.

We store attributes and methods of a class in separated record blocks which are indexed by name of attributes and methods. The root of index is kept at the field of "index 1" in Table 1. Additionally, the instances of this class are grouped below the same index. And its root is stored in the "instance index" field in Table 1.

The storage format of object instance is the same way as the conventional storage structure. One object may be stored in separated records rather than organized it into a single file. By this means, although an object has very complex structure and many layers, it still can be stored continuously or indexed by physical pointer directly. In addition to storing object's attributes, we also keep the CID which the instance belonging to and the association with the other objects in our object storage entities. The storage format of an object is as shown as Table 2. In this format, OID means object identification.

Storage format of object instance:

CID	Record length	OID	Attribute 1, ..., attribute n	Associating field

Format of attribute:

Attribute No.	Data type	Length	Data

Format of associating field:

CID	Associating type	OID

Table 2: Storage format of object instance

3.3 Dynamic data types

Dynamic data type needs to have a specific data structure. We introduce it into our ORDBMS mainly for supporting the inheritance of database table (included function inheritance and data inheritance). And it also gives our ORDBMS certain capability of dynamic schema modification.

Define type flag	Name	TID	Reference times	Index 1, 2	Instance index

Table 3: Storage format of dynamic data type

When a dynamic data structure is used as an attribute of the others, we call this dynamic data structure referenced one time. The reference times field in Table 3 is used to account the referenced times. And the TID means Type IDentification.

In a similar way to the storage format of class, we store attributes and functions in separated record blocks. These records are indexed by the name of functions and attributes. Roots of the two indexes are kept in the definition of dynamic data type. Further more, the instances of the one dynamic data structure are grouped as an index structure. The root of index is kept in the field of instance index in Table 3. All of dynamic data type are grouped as an index and its root is kept in a index directory table of dynamic data types that is named with system directory table.

Clearly, for introducing the capability of nested structure, the capability of description complex object has been greatly enhanced. As we all know. The description one product is always a complex data layer-structured in the engineering application. A product is composed of components. And the components are often composed of some sub-components and parts. In a database operation, whole product data or whole component should be viewed as a unit of storage operation. And this requirement can be easily met in our ORDBMS system. But in the conventional database system, this is the user's work.

4 Generalized B-trees

Indexing structures and object representation are treated separately in spatial database applications. There are three general approaches of indexing structures for extended objects[5]. First of all, the transformation approach is for parameter space indexing. The second one is nonoverlapping native space indexing approach. A k-dimensional data space is partitioned into pairwise disjoint subspaces. These subspaces then are indexed. The third one is the overlapping native space indexing approach. The basic idea of this approach is to hierarchically partition the data space into a manageable number of smaller subspaces. We use the first approach in the system. That is, all entities are described by n vertices in a k-dimensional space as 4 new data types described above. And they are indexed in an n.k-dimensional space with a generalized search trees.

Previous search trees only support range predicates ($<, =, >$), so if you index, say, a geographic information system with B-trees, you can only ask queries like "Find all towns that population < D1". While this query might mean something (e.g. "<" could mean "near of a river", Or "smaller area"), it's not really a natural thing to ask about some area.

Conventional B-trees index can efficiently be satisfied the requirements of relation information layer. So we define a Generalized B-Trees (GBT for short) code that is basing on m-ways search tree structure with the user-defined method. It offers tools for user to stick up new data type, new operators and new query functions on the B-trees codes for satisfying the requirements of complex object information layer.

A GBT is a balanced tree which provides "template" algoritms for navigating the tree structure and modifying the tree structure through node splits and deletes. Like all other (secondary) index trees, the GBT stores (key, RID) pairs in the leaves; the RIDs (record identifiers) point to the corresponding records on the data pages.

Internal nodes contain (predicate, child page pointer) pairs; the predicate evaluates to true for any of the keys contained in or reachable from the associated child page. This captures the essence of a tree-based index structure: a hierarchy of predicates, in which each predicate holds true for all keys stored under it in the hierarchy. We define the GBT to have the following properties:

1. Procedure GBT has parameters: less_than, equal, greater_than, and KT. Where the less_than, equal and greater_than are used to define comparison operators. KT is used to define the data type of the key Ki.
2. It has at most m sub-trees and its root has the format as:

   $A_0, K_1, A_1, K_2, A_2, \ldots, K_m, A_m$

   where $A_i (0 \leq i \leq n \leq m)$ is the pointer that points to its sub-tree
3. K_i is less_than K_{i+1}, $1 \leq i < n$.
4. If the keys in a node N are arranged in their linear order, $K_1, K_2, \ldots, K_m$, then there is associated linear order among the sub-trees, $A_0, A_1, A_2, \ldots, A_{m-1}, A_m$ of N such that:
   - every key in A_0 is less_than K_1,
   - every key in A_m is greater_than K_m, and
   - for $1 \leq i < m$, every key in A_i lies strictly between K_i and K_{i+1}.
5. Sub-tree $A_i (0 \leq i \leq m)$ is a generalized m-way search tree.

The keys K_i in the GBT are not integers like the keys in a B-tree. Instead, a GBT key is a member of a user-defined class, for example, as defined above, POINT, and represents some property that is true of all data items reachable from the pointer associated with the key. To make a GBT work, we just have to figure out, through assigning parameters to it, what to represent in the keys and what to do in the comparison operators .

4.1 User-defined predicates

Because the data types of the keys are varying, the comparison operators "less_than", or "equal" or "greater_than" that were mentioned in the procedure GBT as its parameters are not to be surely the standard predicates "<", "=" nor ">". It may be a set of user-defined predicates to correspond each user-defined data type. When defining a new data type, a set of operator functions for each new type must be defined. For the geographic application, the user-defined elements are defined to implement the comparison operations about the data types POINT, CURVE, AREA and VOLUME in GBT, as listed as in Table 4.

For example, the data type POINT must have several comparison operators as following: "North_of" (to replace the parameter less_than in the GBT), "South_of"

	Comparison Operator	Data type
In B_trees	$<,=,>$	Integer
User-defined	North_of, Same, South_of	POINT
User-defined	Shorter_than, Equal, Longer_than	CURVE
User-defined	Smaller_than, Equal, Larger_than	AREA, VOLUME

Table 4: User-defined methods for geographic application

(to replace the operator greater_than) and so on. To base it upon geography knowledge, we can define comparison operators function "North_of", "Shorter_than" as following: When defining a new data type, a set of operator functions for each new type must be defined.

```
/* two parameters that have data types POINT*/
Function North_of(point_1, point_2: POINT)
{
  /*to suppose that the positive direction of the
  axis $x$ of coordinate system is north */.
  if(point_1.x > point_2.$x$)
  then return (true)
  else return (false)
}

/*two parameters that have data types CURVE*/
Function Shorter_than(line_1, line_2: CURVE)
{
  a = length(line_1);
  b = length(line_2);
  if($a < b$)
  then return(true)
  else return(false)
}

Function length(L:CURVE)
{
  /*to obtain the vertex number of the curve*/
  N = L.n;
  sum = 0;
  for i=1 to N-1
      sum = sum + distance($p_{i+1}$ - $p_i$);
  return(sum);
}
```

When making a index about a key, the GBT first examine the data type of the key. For example, the data type of the key is POINT. Then the GBT selects the comparison operators "North_of", "Same" and "South_of" from the line of the POINT in the table 4. The three comparison operators are used to replace the "less_than", "Equal" and "Greater_than" correspondingly in the m-ways search tree procedure. So the search tree indexes points according to the user-defined functions "North_of".

5 Conclusion

We combine relational DBMS technique with object-oriented technique together and to develop a two layer structure and storage mechanism of object class. It can process the information about complex data type and have the capability of dynamic schema modification. Because of the system directory table structure, the ORDBMS can support any user defined data and any processing function. In one word it can perfectly meet some specific database application demands.

References

[1] Lougie Anderson, etal, Looking for the objects in object-relational DBMSs, ACM SIGPLAN Notes, Vol.32, No.12, 1997, pp 93-102.

[2] Don Chamberlin, Using the new DB2: IBM's object-relational Database system, 1996, Morgan Kaufmann Publishers, Inc.

[3] Hjelsvold, R. and Midtstraum, R., Modeling and Querying proceedings of the 20th VLDB, Santiago, Chile, 1994.

[4] Shi-guang Ju, Visual methods for the software and tools of DBMS, Computer Engineering & science, Vol.20,No.A1, 1998, pp129-132.

[5] Elisa Bertino, and Beng Chin Ooi, The indispensability of dispensable indexes, IEEE Transactions on knowledge and data engineering, vol.11, No.1, 1999 pp17-27.

[6] Michael Stonebraker and Paul Brown, Object-relational DBMSs, Morgan Kaufmann Publishers, 1998.

[7] Anil Kumar,Vassilis J. Tsotras and Christos Faloutsos, Designing Access Methods for Bitemporal Databases. IEEE Transactions on Knowledge and Data Engineering, Vol.10, No.1 1998 pp1-20.

[8] Volker Gaede and Oliver Gunther. Multidimensional access methods, ACM Computing Surveys, 1998, Vol.30 No.2.

[9] J.Hellerstein, J. Naughton, and A.Pfeffer, Generalized search trees for database systems. In proc.21st Intl conference on very large databases, Zurich, Switzerland, pp562-573, Sept. 1995.