The Implementation of a Server for a Distributed Class Algebra Database System

Chen Chun-Ting, Lo Tse-Win, Hsieh Chih-Ming, Hou Kai-Liang, Daniel J. Buehrer

Institute of Computer Science and Information Engineering
National Chung Cheng University, Chiayi 621, Taiwan
dan@cs.ccu.edu.tw

ABSTRACT

We have implemented a class algebra database server with a distributed client/server architecture. This includes a nice object-oriented structure, an easy-to-use interface, an Offset Table that hides object storage, and objects including User objects, Class objects and standard objects. These objects are serialized onto persistent storage. In order to solve the single inheritance problem of the Java language, we have designed an LClass structure that meets the needs of multiple inheritance for class algebra. For improving performance on searches, an Inverted Attribute/Relation Table has been created. Also a clear interface which includes add/get/set/delete/change/invoke methods of classes, relations and attributes, and a class algebra query language has been provided for the client.

1. INTRODUCTION

In the past twenty years, relational database systems have proven to be quite successful for developing traditional business database applications. The rows in relational tables represent individual objects or data. These rows are usually called tuples or records in a relational database. These attributes are the columns of the table.

However, traditional relational database systems have shortcomings for more complex applications, such as engineering design and manufacturing (CAD/CAM), images and graphics databases, scientific databases, geographic information systems and multimedia databases. These newer applications have more complex structures for objects, longer-duration transactions, new data types for storing images or large textual items. Today, with the benefits of object-oriented concepts, object-oriented databases have been proposed to meet the needs of these more complex applications.

In this paper, first we briefly describe the reasons for implementing a class algebra database. Then we introduce some background that is basic to the implementation. This background includes a brief introduction of the Java programming language [1], properties of Java RMI [2] and reflection [3]. In Section 3, we can see how important class algebra [6-8] applications can be in each field of computer science. This paper describes a prototype class algebra database which was implemented as a secure client-server multiple user architecture.

2. CLASS ALGEBRA

Class algebra provides a mapping from normalized class expressions to sets of objects. Sets, which are the "extent" of classes, can be either classical or fuzzy sets, although we have implemented only classical sets. A query is an implicit definition of a set. This set consists of objects that are in the extent of some class.

- Intent (Classes)

  An intent is a normalized class algebra query. Intents have two uses. For an explicit class or relation, the intent gives typing constraints which are used to restrict the assignments that can be made to that explicit class's objects or relation edges. For queries (i.e. implicit classes) and implicit relations, the intents are used to calculate the current value of the extent or the current set of edges in the relation.

- Extent

  For an explicit class, the extent is the set of objects which have been declared to be in that class or one of its subclasses. For a query (i.e. implicit class), we can derive the extent by testing which objects satisfy the query.

Class algebra has the advantages of object-oriented concepts, such as inheritance, reflection, encapsulation, and strongly-typed attributes, methods and so on. A class is a data type of objects. An intent defines the structure and behavior of objects of a particular type. This structure includes attributes, binary relationships and methods (behaviors). An object is an instance of its definition class. Those objects that have been declared to be of class C have attributes, relations and methods which satisfy the constraints given by C's intent. For example, all objects of type Student have a name, a student ID, an advisor, a sex, a set of methods, and so on. It would be best, however, if the student were declared to be of type "Person", and the person's current value were assigned to a new instance of the "Student" subclass. In this way, the person could graduate and be assigned to an Employee object with the original object identifier unchanged.

For object-oriented databases, classes provide the schema of the database. The structure of the objects in the database is described by the class intents. Objects and classes are similar to the traditional databases in that there are two kinds of information in the database, data and schema (metadata). Classes describe the schema of the object-oriented database system. Classes are also objects in our data model. Each class has attributes, relations and methods that are constrained by the intent of that class.

Class algebra is an algebra of classes and their operators, which include the standard Boolean operators, a dot operator for attributes and binary relations, a selection operator, an ISA superclass/subclass relation, and assert/remove operators. Each class expression maps to a unique normalized version, called the intent. The Intent is used as the label of a node in the ISA hierarchy and this is a normalized class algebra class expression's class name. The extent of the class is the set of all objects for which that membership expression returns a "true" value.

A class expression (query expression) is normalized into a Sorted Disjunction Normal Form (SDNF for short) that describes constraints of the corresponding set of objects. The SDNF is an intent for either an explicit class or a query (i.e. implicit class, also called the query class). The explicit class has a list of all members that have been declared to always satisfy that intent regardless of what assignments are made. For an implicit (query) class, the value of the implicit class is computed each time that the implicit class is invoked. The state of an object will change if attributes or relations of that object change. So an object will move from one implicit class to another implicit class when the state of that object changes. Since the state of objects will change, the value of an implicit class possibly changes each time that the corresponding query expression is evaluated.

The binary relations are also either explicit or implicit in class algebra. An explicit relation is stored as a domain class, a range class, a relation name and a set of Oid cross products. An implicit relation is stored using a domain class, a range class, a relation name and the defining SDNF query for the domain and SDNF query for the range. The SDNF queries produce implicit subclasses of the domain and range classes. The implicit relation is evaluated to get an explicit relation which includes an edge from every domain object to every range object.

The main advantage of using class algebra is that it solves the "containment" problem of determining whether or not one class expression produces a subclass of another class expression. Although it may take exponential time to compute the SDNF forms of class algebra expressions, the SDNF forms may be compared for "containment" in linear time. Therefore, both explicit type constraints and implicit query expressions are quickly organized into a ISA hierarchy to help the user locate appropriate objects. Logically equivalent queries can also be detected. Most other query algebras involve variables which make them run either very slowly or, as in the case of non-stratified Prolog programs, cause the containment problem to be uncomputable.

3. SERVER IMPLEMENTATION

With the benefit of the Java programming language, attributes and methods in our database are strongly typed. By means of Java reflection, a user can inspect information that includes modifiers, fields, methods and so on. There is a clear RMI interface for the clients to manipulate the objects, attributes and relations. Also, serialization [12] allows objects to be serialized as an entity into persistent storage. In this section we discuss the implementation of the distributed class algebra database server.

3.1 The Big Picture

Since the architecture of our database is distributed, objects may be located on several servers in the network. A client needs to connect to all servers available on the network. As Figure 3-1 shows, first a client needs to choose a server from the network that has read and write permissions for the client. We call this server a write server. Then, this client connects to the write server, server *A*, and gets machine signatures of other servers available on the network from this server *A*. Information in a machine signature includes a machine name, a TCP/IP address, an URL address, a public key, a random number and an encrypted random number. After connecting to server *A* and receiving machine signatures, the client connects to other machines if needed.

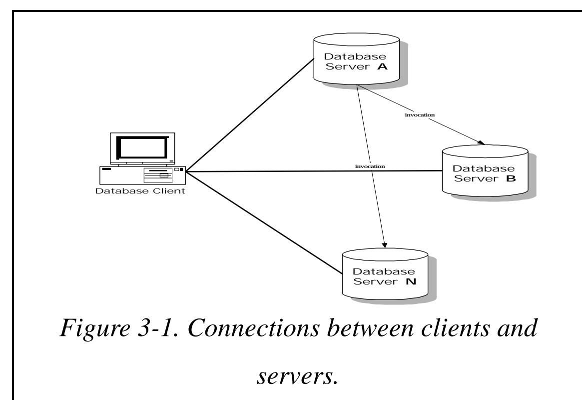


Figure 3-1. Connections between clients and servers.

3.2 Distributed OODB

As we mentioned in the previous section, objects may be distributed across several servers on the network. A user needs to choose a write server for each chosen writable database. The server will prevent multiple users from being assigned write capabilities at the same time. Also a user can connect to several servers that have a read permissions for various databases. Each time when a user connects to a server, the server will return a session to the

user. Our user interface will union the results of queries from the servers that he has connected to, and delegate the updates to the appropriate write servers.

3.3 Server Architecture

Figure 3-2 shows the architecture of a class algebra database server. When a client requests a connection to a database server, the client must create a login process and communicate with the secure login interface of the server. If the user name and the user password are valid, the server creates a remote session for that user and returns a stub [2] of that remote session to the client. The session includes the remote interfaces that the server provides. These interfaces include methods to modify classes, objects, attributes, relations, queries, and so on.

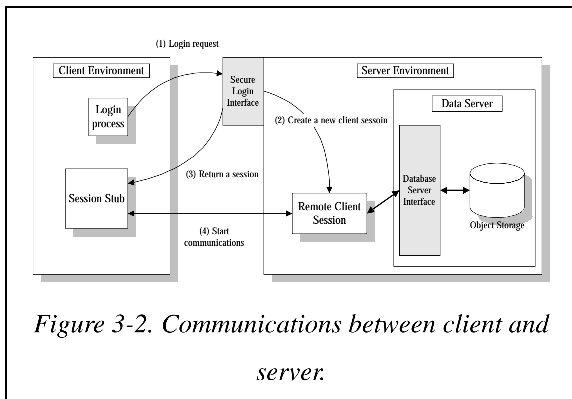


Figure 3-2. Communications between client and server.

3.4 Abstract Class Hierarchy

In this section, we will show an abstract hierarchy (Figure 3-3) to describe the relations between each component of the server machine. This abstract hierarchy is an overview of the server. We do not describe details here. As Figure 3-3 shows, a user can login (connect) to the other machines and then invoke methods on objects from these other server machines. Here, we assume each machine represents a server. A server has many users that can connect to it. A user has classes that he has declared. A class may have superclasses or subclasses. Also, this class may have objects in its extent. Each object can have binary relations, attributes and methods.

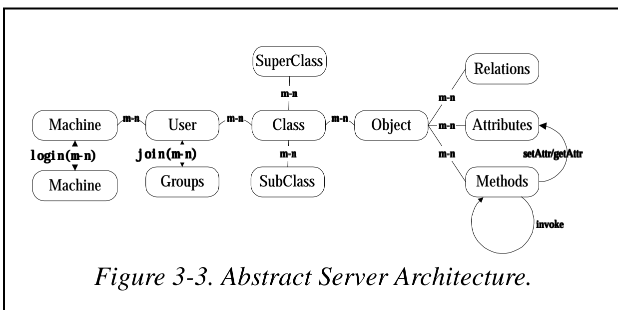


Figure 3-3. Abstract Server Architecture.

3.5 Object Identifier (OID)

Each object in our database has a unique object identifier (OID for short). The object identifier identifies a unique object in a database. If an object was deleted or removed, the unique OID of that object cannot be used by another object. The OID in our data model is composed of three fields, and these fields, are separated by commas in the OID string (as Figure 3-4 shows).

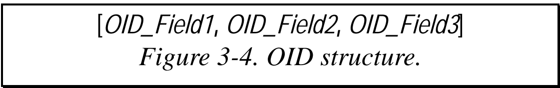
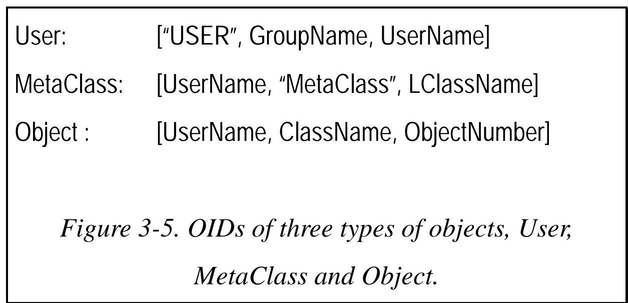

[*OID_Field1*, *OID_Field2*, *OID_Field3*]

Figure 3-4. OID structure.

In our database model, there are three types of objects that must be stored into external object files. They are the User objects, Classes, and the objects which are instances of classes. These types of objects use the same triple structure (Figure 3-4) for their object identifiers, so it is necessary for us to distinguish them from each other.



User:	["USER", GroupName, UserName]
MetaClass:	[UserName, "MetaClass", LClassName]
Object :	[UserName, ClassName, ObjectNumber]

Figure 3-5. OIDs of three types of objects, User, MetaClass and Object.

In a User object, the first field has a tag of "USER". The second field represents the group that the user joined, and the third field stores the name of the user. In a Class object, the first field indicates the owner of this Class and the second field is a "MetaClass" tag. The third field of a Class Oid stores the class name. In an Object, the first field stores the name of the owner of that object, the second field stores class name, which is the declared class of that object, and the third field stores the object number which is generated uniquely by the declared class of the object. (see Fig. 3-5.)

3.6 OidObject Structure

Here we describe the object structure. With the serialization functionality of the Java programming language, we have designed an OidObject structure, which can be serialized/de-serialized into external object files through an input/output stream [12]. So the OidObject is the basic structure that can be serialized/de-serialized to/from object files in the database system.

As we mentioned in the previous section, there are three types of objects that need to be written to external storage. However, the only object that can be written to external storage is an OidObject object. This problem is solved with careful design of the OidObject. We design an OidObject as a capsule that can store all kinds of objects inside it, and then this capsule can be written out to or read from object storage, a disk file, through the ObjectOutputStream [12] mechanism of Java Serialization.

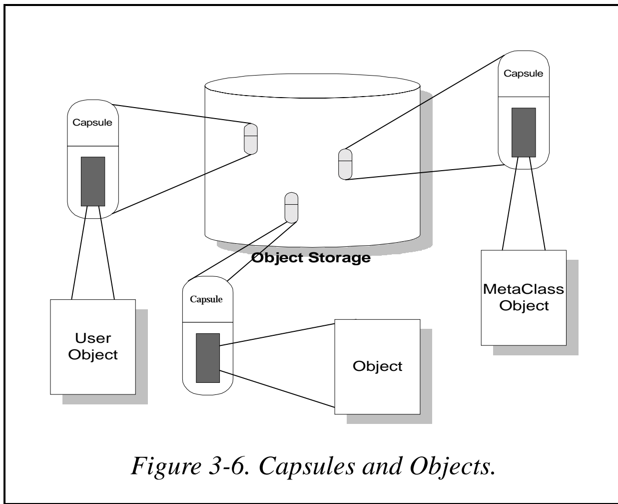


Figure 3-6. Capsules and Objects.

How is this done? In Java, all objects are subclasses of class Java.lang.Object. In the definition of class OidObject, there is a field called "object", which is of type Java.lang.Object. Objects that are subclasses of Java.lang.Object can be assigned to this field. So, User objects, Class objects and non-class objects can be encapsulated into an OidObject. An object that could be written to or read from external storage must implement a Serializable interface [12] or extend a class that is serializable. The OidObject class implements the Serializable interface, and it can be read from Java ObjectInputStream [12] and written to Java ObjectOutputStream.

3.7 User Objects

A User object keeps information and the state of a database user. Here is the information included in the User object:

- User Name
  A User Name represents the owner of the database. He has the capability to assign his method call capabilities (e.g. read/write/delete) to others.

- User Groups
  The names of groups which this user has joined.

- Encrypted Password
  An Encrypted Password is used to check the user's login password. Each time the user requests to change his password, the new password needs to be stored in this field.

- Classes (LClasses)
  "Classes" is a set that records the LClasses (described in the next section) that the user has declared. When the user declares a new class, the server must save this new class in this set. When the user deletes a class, that deleted class and the subclasses which have no other direct superclass, along with their relations, must be removed from this set.

```
public class OidObject implements Serializable {
    private String userName;            // userName.
    private String className;           // which class, ref by ClassTable
    private String objectNumber;        // object id itself, used for OidTable
    private LVersion version;           // object version.
    private Object object;              // keep Real Object.
    /* for optional attributes, pairs of AttributeName:AttributeValue. */
    /* AttributeName and AttributeValue are of type String. */
    private Hashtable optionalAttributes;
    /* for relations (both required/optional) */
    private Hashtable relations;
    private Hashtable inverseRelations;
    private String currentClass;    // must be a subclass of className.
}
```

Figure 3-7. OidObject class.

3.8 MetaClasses

We defined the LClass as follow:

```
public class LClass implements java.io.Serializable {
    private String name;
    private LCatalog attributes;
    private Hashtable methods;
    private HashSet supers;
    private HashSet subs;
    private HashSet instances;
    private HashSet relations;
    private HashSet inverseRelations;
}
```

where the "name" means this class's name, "attributes" is the field names and types, and "methods" as all methods of the class that we can invoke. The "supers" indicates the super classes. Similarly, "subs" indicates the sub classes.

We believe that every user has the right to change his class definitions, and also has the right to keep any data that he has stored into the database. So there is something incompatible between these two ideas. We use a special version control mechanism to balance them. We define two kinds of versions: Catalog Versions and Attribute Versions. Every time that the class owner add a new attribute, then the database system issues an Attribute Version to the new attribute. The Attribute Versions will never repeat.

```
public class LVersion implements java.io.Serializable,
Comparable {
    String owner;
    int verno;
```

```
}
```

The "owner" is a combination of Host and UserID. It is assumed that user names are unique on the net, in the same way as email address. The version number, "verno", is produced by a counter, and it is always increases after every issue of a new version number. Whenever the attribute is updated (no matter whether by add or delete), the Catalog Version will increase, and the Version will pass to all subclasses. So we can know who made the update.

Within the mechanism above, we have built a multi-layered data structure to handle the changes to the catalog:

```
public class LCatalog implements java.io.Serializable {
    private Matrix data;
    private String classname;
}
```

The "data" field is a 2D matrix to record every change of the catalog. The table format looks like this:

Versions	ObjsCounter	AttrsCounter	*Attr1*	*Attr2*	*Attr3*
Ver1	50	3	true	true	true
Ver2	40	2	true		true
Ver3	100	2	true		true
Ver4	137	1			true
Ver5	0	1			true

Figure 3-8. Table of Object Versions.

The "Versions" field indicates the Catalog Version. "ObjsCounter" indicates the number of objects which have been created for this version. "AttrsCounter" indicates the number of attributes for this version. The union of the attributes that we have used to create objects is sorted by Attribute Version.

We force users to only use the newest catalog to save new data. By using the mechanism above, the user can still read his old data.

The "methods" field in LClass is a hash table whose key is the method name, and whose value is a java bytecode file name. The user can write his own method as a class, and upload it to the server, as follows:

```
public class Method {
    Object invoke(Object thisobject, Object[] parameters);
}
```

When the user invokes some object's method, the server will search for the registered java bytecode file, and then load it using a ClassLoader. Then it will call the method "invoke", and pass "thisobject" from the query result, and parameters to the method, then get the return value to send to the user.

For preventing name collisions of methods, the server will add the user name to the class name of the bytecode. Otherwise, because of name collisions among methods, there would be some trouble for Java's ClassLoader.

3.9 Object Storage

- The architecture of this implementation includes an object file and an object offset table. An object file is a logically persistent storage. We can serialize objects into the object file. An object offset table maps an object identifier (OID) to the offset address in the object file.

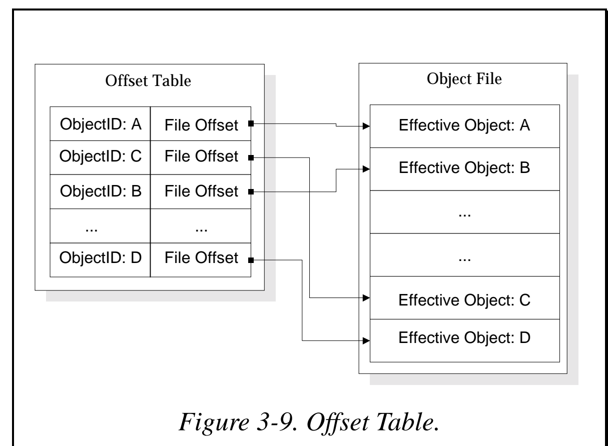


Figure 3-9. Offset Table.

3.10 Attributes

Objects have attributes. Attributes represent an object's properties, states, or other qualities. For example, a car has a color and a length attribute. Here we define attributes by variables of Java classes.

Variables defined in classes are required attributes. Class Algebra allows an object's attribute to be created dynamically. This could be done with Java's inheritance mechanism. For example, when a user wishes to add a new attribute, named "height", to a class, named "Student", the server will add a new class with a variable of "height", and other variables and methods that are inherited from its Student ancestor class. We name this new class "Student1". So class "Student1" has all of the attributes and methods inherited from class "Student" and its ancestors, including the new attribute, "height".

A problem occurs when the user makes a reference to "Student". Since users do not know the inside mechanism of the database server, we have to provide an interface so that the user does not need to know which class, "Student" or "Student1", he has referred to. The approach is to design a ClassMap, which is a hash table that maps a class name to its effective class name, i.e. "Student" to "Student1". Each time that a user accesses a class which is

given by its class name (e.g. "Student"), the class returned (e.g. "Student1") is always its effective class.

Since the Java language has single inheritance, we need a different way of inheriting attributes from a superclass to a subclass. So we create a hash table, called "required attributes", that stores the attributes which are inherited from superclasses. This hash table maps an attribute name to an aggregate structure which contains the attribute name, an attribute type and an attribute value.

Objects can also have optional attributes. An optional attribute is an attribute that has no inherited typing constraints. A hash table was created in order to store the values of these optional attributes. Also this hash table maps an attribute name to an aggregate structure which contains the attribute name, an attribute type and an attribute value.

Attribute editors and attribute viewers are two kinds of interfaces for accessing the value of an attribute. An attribute editor uses the setAttribute interface of a Java Bean. It can change the value of an attribute to a new value. An attribute viewer can only use the getAttribute interface of the Java Bean. It simply returns the value of the attribute. These two kinds of interfaces, the attribute editor and attribute viewer, are provided by the user who defines a new primitive attribute class. For example, the attribute editor for an MPEG-4 object may provide editing methods, such as adding a scene or inserting an object. The attribute viewer for an MPEG-4 object may simply provide a play method that displays the MPEG-4 stream. These new primitive objects may then be added into cells of the user interface's grid.

3.11 Attribute/Relation Inverted Table

Class queries often involve searching for a given value of an attribute or relation. We have designed a mechanism for quickly responding to such queries. The mechanism is an Attribute/Relation Inverted Table. As Figure 4-12 shows, the Attribute/Relation Inverted Table maps an attribute and its value to a set of Oids, where these Oids point to exactly those objects that have that attribute and its corresponding value. Also an Attribute/Relation Inverted Table maps a relation to a set of Oids where these Oids have that value for the relation.

At the time that an object is created, all attribute values (both required and otional), and relation values must be added into an Attribute/Relation Inverted Table. When an attribute value or a relation value of an object is changed, the corresponding Oid set of that object must be modified. Also the modification of the Attribute/Relation Inverted Table must be done when an object is removed or deleted.

3.12 Relations

In our database model, there are two kinds of relations. They are explicit relations and implicit relations. An explicit relation includes a domain class, a range class, a relation name and a set of Oid cross-products, where the domain the and the range class are LClasses. Objects that are in the left side of a cross-product must satisfy the intent of the domain class and objects that are in the right side of a cross-product must satisfy the intent of the range class.

An implicit relation includes a query expression of a domain class, a query expression of a range class and a relation name. Objects that return a true value for the query expression of the domain class will be selected, and objects that return a true value for the query expression of the range class will also be selected. The implicit relation represents the cross product of these two sets. An implicit relation is usually made up by unioning subrelations which all have the same name, but which represent different cases of the relation.

The major difference between implicit relations and explicit relations is that the result of implicit relations is computed each time from its intent. If the current state of database is different from previous state, the result of a class expression may be different.

As we mentioned previously, an implicit relation in a LClasses contains a query expression of the domain class, a query expression of the range class, and a relation name. An explicit relation contains a domain class, a range class, a relation name and a set of pairs of Oids. Also the inverses of implicit or explicit relations are kept in LClasses. These inverse relations are good for inverse queries from range classes to domain classes. Explicit relations are also stored in the objects themselves. Relations in objects are all explicit ones, and they are recorded in the form of a set of range Oids.

3.13 Method Invocation

We use Java's strongly-typed methods to implement behaviors. That is, methods in the Java language have types. The user can call these methods if they have a public access tag. So users can invoke methods from objects. We have designed a secure invocation approach for the database server. The approach is that when a user wants to invoke a method on some objects, the user will communicate to the Secure Invocation Interface. That user must specify the method name, arguments and Oids, on which the invocation will occur. Then the Secure Invocation Interface will check whether this user has permission to invoke the method on these objects. If the check is passed, the Secure Invocation Interface will delegate invocations to the Generic Invocation Interface. Otherwise an error object that indicates an illegal access will be returned. After the check is passed, invocations will be continued by the Generic Invocation Interface. The Generic Invocation Interface will then invoke methods from those specified objects. Also error objects, that

indicate objects on which the invocations have failed, will be returned to the user from the Generic Invocation layer.

There are objects that may be located on some other servers on the network. At this time, the write server will delegate invocations to the other servers. So servers in our database model provide an invocation interface to other servers.

3.14 The Query Language and Interfaces

The context free grammar of the class algebra query language of the database server is given below. We use the Java-based tools JLex [4] and CUP [5] to parse these queries.

```
<classexpr> ::= <classexpr> '+' <classexpr>    //union
              | <classexpr> '~' <classexpr>    //not (same as &~)
              | <classexpr> '-' <classexpr>    //difference
              | <classexpr> '*' <classexpr>    //intersection
              | '('<classexpr>')'
              | <dottedExpr>
              | <rangeList>
              | <attrName>
<dottedExpr> ::= <optHome> <dottedRelns> <optAttr>
<optHome> ::= 'home.' | <empty>
<dottedRelns> ::= <pathPart> | <pathPart> '.' <dottedRelns>
<optAttr> ::= <attrExpr> | <empty>
<attrExpr> ::= <aggrFcn>'(' <classexpr> ')'
              | <attrName>
<aggrFcn> ::= 'cnt' | 'avg' | 'sum' | 'std' | 'min' | 'max'
<arithop> ::= '+' | '-' | '*' | '/'
<pathPart> ::= 'classes'          //return classes declared by the user
              | 'extent'          //return a set of oids of the class.
              | <relnName>
              | <relnName>'{' <wherecond> '}'
              | <relnName> <RangeList> <optWhereCond>
<optWhereCond> ::= <wherecond> | <empty>
<relnName> ::= <identifier>
<wherecond> ::=
     <wherecond> '|' <wherecond>    //"or"
     | <wherecond> '&' <wherecond>  //"and"
     | '~'<wherecond>    //true if <wherecond> is not true
     | '-'<wherecond>    //true if <wherecond> is false else true
     | <predicate>       // may be 3-valued; true/false/unknown
     | <classexpr> ('~' | '-' | ) ('in' | 'has' | '=') <classexpr>
     | <amount> '(' <classexpr> ',' <wherecond> ')'
<amount> ::= <comparision op> <Integer> ('%' | <empty>)
   | all | most | some | no | always | usually | sometimes | never
<className> ::= <identifier>
<attrName> ::= <identifier>
<number> ::= <Integer> | <Float> | <Double> | <Long>
<Date> ::= <integer> '/' <integer> '/' <integer>
```

A class algebra expression consists of predicates, selection conditions (where conditions) and operators which include dotted operators and set operators (such as union, intersection and difference). A "where" condition consists of Boolean operators applied to "A in Y", "A has Y", and "A = Y" predicates. We define "A in Y", "A has Y", and "A = Y" as:

- "A in Y" means that all values of the attribute or relation named A must have a true value for "this.Y". For example, a class expression,

  classes[Student].extent{ age in [30:inf] },

  means that the student whose age is greater than thirty will be selected.

- "A has Y" means that all values of Y is contained in all values for the attribute or relation named A. For example, a class expression,

  classes[Politician].extent{ nationalities has ["Taiwan", "USA"] },

  means that the politician whose nationalities include Taiwan and USA will be selected.

- "A = Y" is logically equivalent to "(A in Y) & (A has Y)".

In the following, we will give some more examples of queries to explain the usage of class algebra relations and where conditions.

- A database user, called John, who wants to get the extent, which is a set of oids, of the "Student" class that John has defined would give the following query expression (This query will return the oids of objects in John's "Student" class.):

  user[John].classes[Student].extent

- Each of the students whose birthday is between 3/20/1977 and 5/15/1990.

  classes[Student].extent{birthday in ["3/20/1977":"5/15/1990"]}

- Each of the students who takes at least 7 courses.

  classes[Student].extent{cnt(taking) in [7:inf]}

- Each of the students whose the sum of the credit hours is at least 21.

  classes[Student].extent{sum(taking.creditHours) in [21:inf]}

Here we give an example which are quoted from the book of Elmasri and Navathe [14]. We will explain how to use the SQL language and Class Algebra query expressions to get the answers of this example.

Q: **SELECT** FIRSTNAME, LASTNAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME = "RESEARCH" **AND** DNUMBER = DNO

is equivalent to:

```
q := home.classes[Employee].extent
        {worksAt has home.classes[Department].extent{dname =
        "Research"}}
q.firstname, q.lastname, q.address
```

or

```
q := classes[Department].extent{ dname =
        "Research" }.inv(worksAt)
q.firstname, q.lastname, q.address
```

where there exists a relation, named worksAt, between Employee and Department.

The query expression classes[Department].extent{ dname = "Research" } will return a set of Department objects which have an *dname* attribute and the value of this attribute is "Research". The "inv(worksAt)" relation is the inverse of the relation relating workers to their department. In this query, it gives all of the workers in the research department.

Notice that the class algebra queries do not need to join artificial keys like DNO in the employee class, but simply follow a "worksAt" relation to find the departments at which the user works. This is usually faster than joining the DEPARTMENT.DNUMBER and EMPLOYEE.DNO keys. Using multiple names like DNUMBER and DNO is also confusing for programmers.

4. CONCLUSIONS AND FUTURE WORK

We have introduced the object storage, Offset Table, Oid structure, LClass structure, Inverted Attribute/Relation Table and interfaces, including add/get/set/delete/ change/invoke methods of classes, relations and attributes, and a class algebra query language for a class algebra distributed database server. We also have provided several examples to show the power of the class algebra query language.

The performance of the object storage system we designed is not very efficient. So we need to provide an efficient approach to increase the utilization and performance of the object storage system. We can use a data structure, such as a B+ Tree, as the index system of the Offset Table, and we can apply an object buffering system on the memory management system. With this buffering system, the objects are not read one by one, but page by page, where a page may contain several objects. This can decrease the disk access time and improve the performance of the object storage.

The assignments that include add/get/set/delete/ change/invoke classes, relations and attributes, are separated as methods from the class algebra query language. We should integrate these assignments into the class algebra query language to provide a compact interface for the users.

Acknowledgments: We would like to thank the National Science Council of Taiwan for their support of the related projects: NSC 892218E194009 A Voice XML 0.9 Development System and Evaluation System, NSC 892213E194006 A Secure, Distributed Java Database System with Voice Input, NSC 872213E194005 Ontology Reasoning System, NSC 862213E194004 An Object-Oriented Java-Based Database System

5. REFERENCES

[1] Sun Microsystems Inc. Java Software Homepage. http://java.sun.com

[2] Java Remote Method Invocation (RMI): Java 2 SDK. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html

[3] Java Reflection: Java 2 SDK. http://java.sun.com/products/jdk/1.2/docs/guide/reflection/index.html

[4] Elliot Berk, "A Lexical Analyzer Generator for Java™", version 1.2, May 5, 1997. Department of Computer Science, Princeton University. http://www.cs.princeton.edu/~appel/modern/java/JLex

[5] Scott E. Hudson, "CUP: LALR Parser Generator for Java™. Graphics Visualization and Usability Center Georgia Institute of Technology", Version 0.10j, July, 1999, modified by Frank Flannery, C. Scott Ananian, Dan Wang with advice from Andrew W. Appel.

[6] Buehrer, Daniel. J., "From Interval Probability Theory to Computable First-Order Logic and Beyound", Proc. of IEEE World Congress on Computational Intelligence, Orlando, Fla, Jun 26-July 2, 1994, pp.1428-1433.

[7] Buehrer, Daniel J., "An Object-Oriented Class Algebra", Journal of Computing and Information, Proc. of Seventh International Conference of Computing and Information (ICCI'95), Trent University, Peterborough, Ontario, Canada, July 5-8, 1995, pp. 669-685.

[8] Buehrer, Daniel J., "Class Algebra as a Description Logic", International Description Logic Workshop, Boston, AAAI Press Tech. Report WS-96-05, Nov. 2-4, 1996, pp. 92-96.

[9] Lee Jing-Ming., "A Java Object-Oriented Database Server", MS Thesis, Computer Science and Information Engineering June 1999 Chia-Yi Taiwan 62107, Republic of China.

[10] Java Virtual Machine: JavaTM Standard Edition Platform Documentation. http://java.sun.com/docs/books/vmspec/index.html

[11] Java Serialization: Java 2 SDK http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html

[12] Class Loader: Java 2 SDK http://java.sun.com/products/jdk/1.2/docs/api/java/lang/ClassLoader.html

[13] Ramez Elmasri and Shamkant B. Navathe, "Fundamentals of Database Systems", Addison-Wesley, 1994.