Building Scalable View Module of Object-Oriented Database

Haeng-Kon Kim[1] and Hyun Yeo[2]

[1] School of Information Technology, Catholic University of Daegu, Kyungbuk, 712-702, Korea
hangkon@cu.ac.kr
[2] Dept. of Information and Communication Engineering, Sunchon National University, Suncheon, Jeollanam-do, Republic of Korea
yhyun@sunchon.ac.kr

Abstract. Many researchers and developers have studied object-oriented relational database management systems (ORDBMS) in the past ten or so years, few have published their results that reveal the inside workings of an ORDBMS. Leading database software companies integrated object-oriented features in their DBMS products only recently. These companies do not make the technical core of their products public. Most academic researchers, on the other hand, have worked on ideas, methodologies and analysis of ORDBMS, but few have shown the database engine of a working system. This paper presents a propotype ORDBMS engine that supports objects in databases, including user-defined types, inheritance, and polymorphous method invocaiton. Although a prototype, it is implemented in Java, fully functional and can be extended should additional modules be added in the future. The system is composed of three major components: the query-command module, the view module, and the database module.

Keywords: Object-Oriented Relational Database Management, User Friendly Interface, Inheritance, View Module, Searching Engine, Self-diagnosing, Agent, Ubiquitous computing, CBE (Common Base Event), Crop Production Agent Systems.

1 Introduction

Object-oriented database, as one of the latest developments in the database area, has been widely studied in the past decade [2-16]. One of the primary methods used today for object-oriented databases is the object relational model, which is an extension of the relational model. In object relational model, attributes of objects can be of user-defined types such as chunk/stream, set, nested/reference attributes, as well as the primitive types. In addition, new data types and tables can be defined and created by inheriting from existing ones that also supports polymorphous method invocation. As an example, let's say that we want to create audio and image associated with person objects in a human resource database. Using the object relational model, the person table can be defined as

S. Omatu et al. (eds.), *Distributed Computing and Artificial Intelligence, 11th International Conference*, Advances in Intelligent Systems and Computing 290,
DOI: 10.1007/978-3-319-07593-8_35, © Springer International Publishing Switzerland 2014

```
CREATE TYPE media FROM chunk (length: integer);
  CREATE TYPE audio FROM media;
  CREATE METHOD play FROM
  { /home/media/audio/play.class } INTO audio;
  CREATE TYPE image FROM media;
  CREATE METHOD display FROM
  { /home/media/image/display.class } INTO image;
CREATE TABLE person (ssn: text, name: text, salary: num-
ber, face:image, voice: audio);
```

In this design, chuck is a primitive type (a stream of bytes) and the play/display methods are Java bytecode in the specified directory. Note that both the types audio and image inherit from media type, each having its own method for playing the audio or displaying the image. Now, we can query the database like this:

```
SELECT name, face.display( ) FROM person
WHERE salary < 30000 and voice.length < 5K;
```

This example illustrates some of the important aspects of object relational databases, such as creation of new types, inheritance, and method calls.

In this paper, we investigate problems in the design of database management systems in our research endeavor, object-oriented database management systems (ORDBMS) in particular, and present a database engine that support OO databases. Object-relational database management system is a database system that extends the capability of relational database management system (RDBMS) to provide richer data type system and object orientation. The extension attempts to preserve the relational foundation of RDBMS while providing more modeling power. During the past several years we had built some prototypes using C++ as the implementation language but it appeared that some of the features that were supposed to be supported by ORDBMS were quite difficult to realize using C++. Problems particularly troublesome were the handling of pointers in nested and reference attributes as well as methods applied to multimedia objects in the database. In early 1998, we switched to Java [1] in our implementation. It turns out that Java and ORDBMS fit extremely well. Java provides straightforward support for multimedia object handling, secure object referencing, and elegant inheritance mechanism. We have designed an ORDBMS engine and implement it in Java. A simple user interface was also developed for testing the database engine.

The architecture of the ORDBMS consists of three major components:

- Database module: providing the main concepts and related object classes needed to deal with the manipulation of database schema and low-level data storing and retrieval.
- Query-command module: processing user's query and other actions, providing mechanism to support commands at the language level (like "Statement" and "ExecuteQuery" in JDBC).

- View module: providing the main concepts and related object classes used to support the mechanism of attribute referencing, expression composition, projection, filtering, and joining.

The overall architecture is shown in Fig. 1.

We will describe each of the three components in the next sections, along with the Java API class hierarchies. In this paper, class names are in *italic* and start with capital letters.

2 Database Module

This is the lowest level component of the ORDBMS, i.e. it is the module directly interacts with the database. It processes the database schema and stores the metadata in internal tables, which are also known as "catalogs". It also manipulates the storage and retrieval of data in the database. The database module provides the functionality to support various kinds of data types in a uniform way. These data types include reference, set, nested field, chunk (raw stream of bytes for image, audio and video data), and method, in addition to the built-in types such as integer, float, boolean, text, and chunk. The major object classes in this part of the ORDBMS are: *Database, DataType, MethodInfo, TableDef, FieldDef, Record, Attribute, RecordSet, Table, TableRecord*, and *Field*. These Java classes (and some classes in other modules) are organized in several groups whose hierarchies are as shown in Fig. 2 below. These class hierarchies are only a part of the architecture of our system.

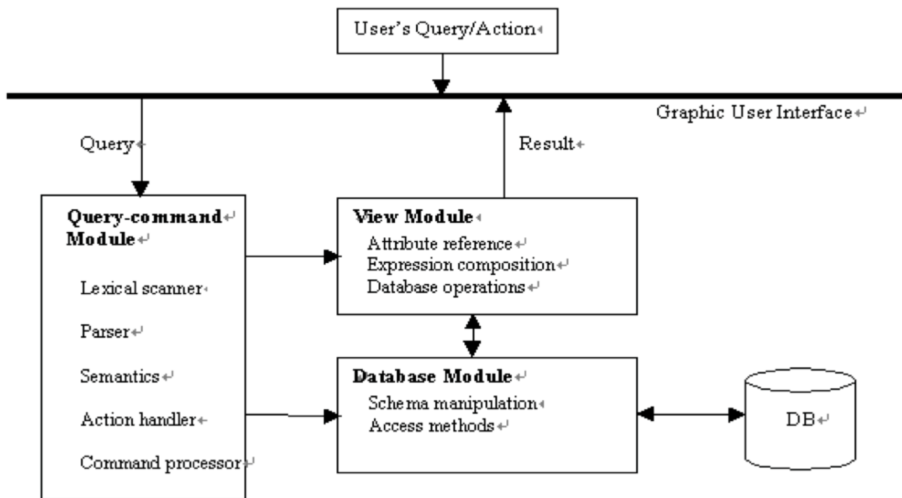

Fig. 1. Overall architecture of the ORDBMS

In the next several subsections, we discuss the major classes of the Database module.

2.1 The *Database* Class

This is the core class that represents the logical concept of a database system. The *Database* object handles the schema; persistently keeps, manipulates, and provides information about the database, and controls the internal works of the system such as file access control. For example, when the user adds a new type to a database, the information of the new type is stored and all of its references are resolved within the *Database* object. The main function of the methods in this class includes:

- Create a new or open an existing database. If an existing database is opened, the *Database* object loads the schema information (definitions of tables, fields, and methods).
- Add and drop table/field/method definitions.
- Close a database.
- Read and write database files.

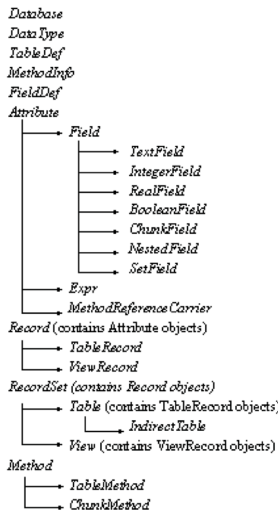

Fig. 2. Class hierarchy

The *Database* object uses *TebleDef* and *FieldDef* objects to maintain the schema information for object values (expressions, constants, operators, etc.), classes for records of database tables, and classes for method attributes. Because a user-defined type may describe objects with multiple attributes, it is treated the same way as tables in the database.

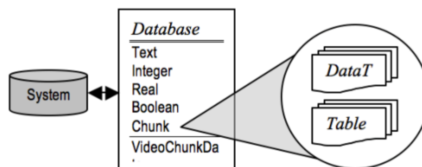

Fig. 3. DataType and TableDef objects inside Database

2.2 The *DataType* Class

In our system, all data types (built-in or user-defined) are represented using DataType objects. A DataType object contains a type name and a reference to the parent type, which may be null. Treating data types uniformly using the same discipline makes the design clean and more manageable.

When a *Database* object is created, it is initially empty. Once the initialization process is done, a set of *DataType* objects are created automatically within the *Database* object to present the built-in data types such as integer, real, text, boolean, chunk, etc. The parent type of each of the built-in types is null. Later, when a user-defined type is created, it is processed and maintained by the *Database* object. There are two kinds of user-defined data types. One is a stream of chunk type. A *DataType* object can directly represent this kind of type because it does not contain more information than that in a *DataType* object. The other kind of data type is structured type, such as a person or an employee. It should be clear that a structured type is simply a table definition in a database. An elegant way to support such structured types is to derive a new class (*TableDef*) from the *DataType* class.

2.3 Table Definition and Tables

Whenever the user defines a type, he/she is defining a table structure. A *TableDef* object contains more information than its parent *DataType*. It represents the definition of a table, which includes the descriptions of all the fields (attributes) of the table. The field definitions are represented by *FieldDef* objects, which will be described in the next section. Additional information stored in a *TableDef* object includes the location on the disk to store the data and a set of field definitions. Conceptually, the user always define a type by creating a *TableDef* object and added it to the *Database* object, which maintains the type like any other *TableDef* objects. Persistence of these types is also guaranteed by *Database*. Fig. 3 shows the *DataType* and *TableDef* objects within the *Database*.

Once a *TableDef* object and the relevant FieldDef objects are created and added to *Database*, actual data can be stored in a Table object according to the definition of the table. A *Table* object acts as an agent between the user and the data storage, as shown in Fig. 2. It handles the overall operations of a table, such as adding or deleting a record. It also controls the movement of the current record. Because the complexity of the kinds of records in tables, storing and retrieval of various kinds of records should be left to other classes specifically designed to handle records and their fields.

2.4 Field Definition and Fields

When we create a table definition, we must define its field definitions first. A *FieldDef* object can be created for this purpose that can be easily added to the *TableDef* object. The important information of a field definition includes the name of the field, its type (reference to a *DataType* object), size, and kind. In our system, there are three kinds of fields: a simple type, a nested or a set type. A nested field is very

similar to having a structure variable inside another structure variable in C++ or Java. When a nested field is accessed, we go though each individual element in the field. This will lead to accessing to another nested field. This process continues (recursively) until all elements in the "current working field" are of simple types. For set fields, it is like a table nested inside another table, because what we obtain for accessing a set field is a table (set of records). Fig. 4 shows an example of nested and set fields.

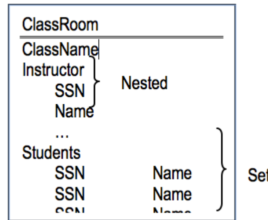

Fig. 4. An example of nested and set fields

To keep track of the data type of the fields is the task of *FieldDef* objects. All built-in types and user-defined stream/chunk types are simple types; all other user-defined types are either nested or set fields.

As stated in the previous section, a *Table* object can be created for storing and retrieval of data after *TableDef* and relevant *FieldDef* objects are added to the schema. The *Table* object only handles the overall operations of a table; it leaves the specific operations to the records in the table to the classes *TableRecord* and *Field*. A *TableRecord* object is given the information about the type and size of the record, and performs the required action to it. Also, it provides a pathway to invoke methods associated with the record.

When a *TableRecord* object performs operations to a record, it also relies on another object (the *Field* object) to deal with individual fields. The *Field* class defines the interface for the *TableRecord* object working with the fields in the table. For example, when a *TableRecord* reads data from storage and brings it to memory, it will repeatedly call the load() method of each *Field* object to retrieve and interpret the data.

Because the fields may be of many different types, we choose to make the *Field* class an interface for *TableRecord* to interact with the fields. The *TableRecord* object does not need to care about the field it is dealing with because all fields have the same interface. For the various kinds of types, we create a set of field derivatives (*TextField*, *ChunkField*, *nestedField*, *SetField*, etc., as shown in Fig. 2) that are derived from the *Field* class. When a new data type is created for a field, the *Field* and *TableRecord* classes need no change.

The *ChunkField* class represents a chunk of data, normally BLOB (Binary large Object) or CLOB (Character large Object), as found in many commercial DBMS. The data of *ChunkField* is variant in size and generally very large. As do many commercial DBMS, we use external files to store the filed data. Each such file is given a unique name by combining file name with the table name. It is also given a unique ID that is stored in the record of the table.

The *NestedField* class represents a field of a structured data type that contains other types. Because the structured data type is also a *TableDef*, the *NestedField* object maintains an internal *Table* object for the database table that is automatically opened for access when the nested field is being accessed. Whenever the value of the table is accessed, the record (*TableRecord* object) of the internal table is returned. From then on, the DBMS programmer can go further to the deeper fields in the same manner. A nested field maintains a pointer pointing to a record by using the OID (Object Identifier) of the record. The record ID maps to the location of the record in the data storage. Fig. 5 illustrates how *NestedField* works with OID.

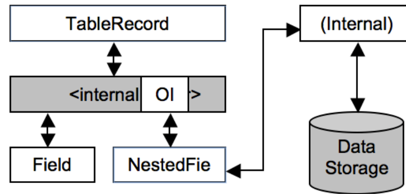

Fig. 5. Access to a nested field using OID

The *SetField* class is designed to handle a field that may contain a table. That is, the value of this field is a table (set of records). For example, we may have a table called Student that contains all students. Then, we can create a table called ClassRoom to keep track of classrooms. This table has a set field called "students" that contains the set of students who attend the class in the particular classroom. This subtable obtained from this field may not contain all the students in the Student table. Moreover, the sub-tables may have duplicates because a student may attend multiple classes. For this reason, we maintain a list of pointers pointing to the original data table. The list of pointers is by itself also a table. So, we create another class called *IndirectTable* that is derived from the *Table* class. An *IndirectTable* object is treated just like any other *Table* object. The only thing special about *IndirectTable* is that it handles pointers.

2.5 Method Definition and Method

ODBMS allows users to create and associate methods with user-defined data types. When a method is created for a type, a method_def object is created, which contains the name of the method, its return type and kind (nested, set, or simple), and the method code itself in the form of a Method object. It is then added to a DataType or a TableDef object the method belongs to. There are two kinds of methods: table method and chunk method. A table method is a method that works on the record basis, and is always associated with a TableDef. When such a method is called, there is always a current table record ready to be accessed for the method. A chunk method works on a byte stream field. It is associated with a user-defined stream/chunk data type. This kind of methods interprets the byte stream as the user defines, mostly for images, audio, or video streams.

Two classes are provided for the two kinds of methods: *Tablemethod* and *ChunkMethod*. They are derived from the *Method* class. The *Method* class, which is an abstract class, defines only the basic interface for a method definition implemented by the DBMS programmer. *TableMethod* class defines more specific interface needed to work on the record basis, whereas *ChunkMethod* defines an interface for working with byte stream. With this design, a method_def object can contain (refer to) a method by having only a reference to a *Method* object, which is a general form of both *TableMethod* and *ChunkMethod*.

The DBMS programmer can create a method by creating a Java class that inherits from with *TableMethod* or *Chunkmethod*, but not from the *Method* class (it is abstract). When the Java class file is created, the DBMS programmer can ask the *db_scheme* object to associate it with a particular table or a data type. Instantiation of an object for such a method (so that the method can be called) is done through Java's dynamic class loading.

3 View Module

The second layer of the ORDBMS is the View module. It provides all the necessary classes so that the ORDBMS programmer can create objects corresponding to the various components in the SQL query issued by the user and constructs an execution plan.

As we have known in relational databases, a view is a result of an SQL query, which is also a table. But may DBMS do not store an actual table for the view; rather, it stores the SQL query itself. When we activate a view, the SQL query is fetched and executed and the result is returned. In this sense, a view is just an SQL query. The data source, from which the data is retrieved for an SQL query, can be either tables (data are actually stored) or other SQL queries (hence nested views).

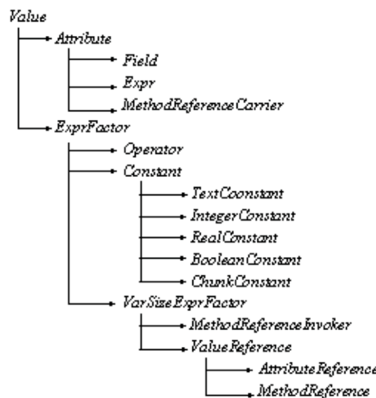

Fig. 6. The *Value* class hierarchy for values in SQL queries

The View module contains classes that handle "values" in an SQL query, including expressions (that in turn contains constants, operators, attribute references, methods, etc. The class hierarchy is shown below in Fig. 6.

3.1 The *View* Class

The *View* class is designed to represent an SQL statement, which in general contains three major parts: an expression list (or field list), a data source list, and a condition. These three parts correspond to the SELECT, FROM, and WHERE clauses of an SQL statement. The condition in the WHERE clause is just an expression but it must be a boolean.

A *View* object does not have its own data; rather, it relies on the data source that may be *Tables* or other *Views*. It creates a temporary table containing the positions (or OIDs) of the records in the *Table/View* data source that satisfy the condition. These positions are generated by applying the condition to each record while the *View* object iterates through the record set. Once the temporary table is generated, the *View* object just uses it as the underlying data source. Each *View* object maintains an internal "current record pointer" that points to a row in the temporary table. This row is by itself another pointer. Hence nested views can be easily handled as a pointer chain. The lowest level of View has pointers pointing to the actual records in the data source. This implementation of *View* using pointer chain is called "record reflection" mechanism that provides some benefits. First, there is no need for large storage space to generate a *View*. Second, it is fast to access the actual data because only positions belong to the *View* are kept. Third, any updates made to the underlying record sets are reflected through the *View* immediately. For example, if a underlying record is deleted, we simply break the reflection chain by null the pointers in the View's temporary table, as shown in Fig. 7.

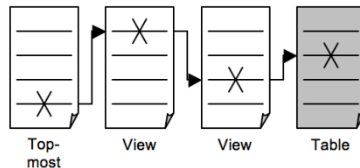

Fig. 7. Deletion of an underlying record in a View via record reflection

3.2 The *ViewRecord* and *Expr* Classes

ViewRecord is designed pretty much similar to *TableRecord*, they both represent records in a table. The difference is that *ViewRecord* objects do not provide data storing capability so that view is always read-only because the result of a view can come from many places (from expressions, join of tables, other views, etc.). The main function of *ViewRecord* is to manage the data obtained from "somewhere" to yield results of the expressions in a list the view is maintaining. In fact, the implementation of

View has an internal *Table* object to work with the temporary table, whereas the *ViewRecord* does not deal with the temporary table at all. The *ViewRecord* is used primarily for simplifying the expression validation process to be consistent with the process handled by *TableRecord*.

An expression can be anything that yields to a value. It can be a constant, a field, a method call that returns a result, or a combination of these. The classes in the Value hierarchy (see Fig. 6) provide all the necessary methods to evaluate an expression in a uniform way. We will discuss several of these classes that are particularly important for the object-oriented DBMS, including AttributeReference, *MethodReference* and *MethodReferenceInvoker*.

3.3 The AttributeReference Class

Recall that fields in an expression may come from different data sources. Each of the data sources has its own internal structure. In order to refer to fields in a uniform way, we use references. One of the commonly used fields is attribute that is represented by *AttributeReference*. Because the object referenced may be some other *Attribute Reference* or *MethodReference*, the getAttribute method of this class recursively calls itself until an *Attribute* object is reached. From the user's point of view, there is no need to go through the reference chain to get the attribute; rather, the user just calls getAttribute and get the target attribute. Using reference also eliminates the need to distinguish field and expression that are encapsulated in the *AttributeReference* object.

3.4 The MethodReference and Method Reference Invoker Classes

MethodReference works in the same ways as *AttributeReference* in the sense that it can also go to several levels due to, for example, inheritance. It refers to a record (Callable record) of which the method is called:

```
public method MethodReference extends Valuereference {
.....
  public Object call(Object[] args) throws Exception {
    try {
      Callable caller = di-
rect ?((RecordSet)source).getRecord():(Callable)((ExprFac
tor)source).getValue();
      return caller.call(MethodName, args);
    } ....
  }
}
```

To certain degree, it is like Class.getMethod() in Java, which gives you a reference to a method. When actually calling a method (may or may not have parameters), we use a *MethodReferenceInvoker* object that encapsulates a *MethodReference* object with a set of parameters (maybe an empty set):

```
public method MethodReferenceInvoker extends Exprfactor {
  private MethodReference mref;
  private Object[] realArgs;
  ......
  public Object getValue() throw Exception {
    return mref.call(getArgs());
  }
  ......
}
```

The following figure shows the relationships of the classes related to the concept of reference.

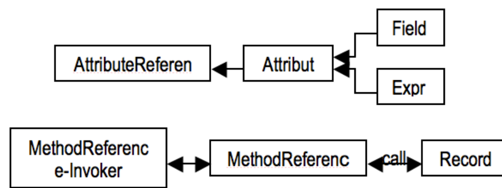

Fig. 8. Attribute and method references

4 Query-Command Module

The highest layer of the database engine is the Query-Command layer that processes SQL-like commands. Three kinds of user input are allowed:

- DDL (Data Definition Language) commands: These commands involve database tables at the schema level. The classes that implement these commands return nothing but will throw an exception if the desired operation fails. These commands are the CERATE command that defines a schema of a table and the DROP command that removes a table from the database.
- DML (Data Manipulation Language) commands: These commands are for accessing the database, including SELECT, INSERT, UPDATE DELETE, etc. Nested queries are supported by the mechanism of view.
- System commands: The user can use these commands to find out the metadata about the database tables. They are the LIST command that displays the types and tables in the database, and the DISPLAY command that displays the internal catalog information about the schema.

This module supports a modified version of the SQL syntax using an LL(1) parser, which is driven by a set of grammar rules defining the syntax. This part of the ORDBMS is responsible for processing user's SQL-like queries and commands. We designed a *CmdProc* class to be a center place where all semantic actions for handling an SQL command are triggered, as shown in Fig. 9.

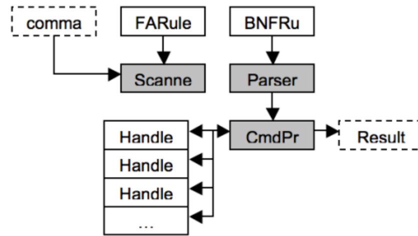

Fig. 9. The architecture of SQL processing

The SQL-like syntax includes all the CREATE, DROP, SELECT, UPDATE, DELETE, etc. statements. In addition, it uses the traditional dot-notation to access or refer to fields and methods of tables. Here are some examples:

```
supervisor.address.city
emp.IncreaseSalary(0.05)
student.getDepartment().chair.name
```

The ODBMS allows a nested field to be treated as a whole. For example, it can be assigned to the field of another object at once. Because we use internal pointers for nested fields, this task is quite straightforward. We use the reserved word ref to be the reference operator in the SQL statement. For example, assume that we have a table called Order that has a field called cust that is of the type Customer. We can write the following SQL statement to insert into the Order table a record that has order_no 0002 by the customer who has placed an order of order_no 0001:

```
INSERT INTO Order(order_no, cust)
VALUES ('0002', ref(SELECT cust FROM Customer WHERE or-
der_no = '0001');
```

Moreover, the ref operator can also be used to reference to an object or a record of a table:

```
UPDATE Order
SET cust = ref(SELECT * FROM Customer WHERE ssn = '123-
45-6789')
WHERE order_no = '0002';
```

The only requirement of the ref operator is that the sub-query to must yield one record.

Nested tables are handled in a similar way, except that it uses a the operator rather than ref operator, as shown in the following example:

```
SELECT *FROM the(SELECT students FROM ClassRoom WHERE in-
structor = 'Dr. Berry');
```

There are some other syntax details for the testing SQL-like query language.

The *CmdProc* class also provides methods to shield the details of processing an SQL statement so that the DBMS programmer can directly pass the SQL statement as a string parameter to the method to carry out the execution, very much like the ExecuteSQLStatement method in Java JDBC.

5 Experimental Examples

To show how the Java ODBMS API works, we creates a "shell" ODBMS with a simple GUI. The ODBMS and all the Java classes in the API are written for JVM 1.2 or JVM 1.1.7 with JFC support. This shell ODBMS simply receives an SQL command from the user and passes it on to the command processor. The result (if any) of the command is then displayed. In the following Fig. 10, we show a few snapshots to illustrate user-defined types, table inheritance, and polymorphous method call. The explanations are given along with the snapshots.

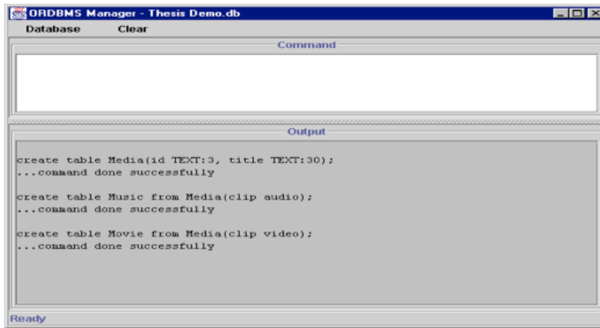

Fig. 10. Three tables: `Media`, `Music`, and `Video` are created. `Media` is the base class of the other two. The derived tables just add only additional field(s) they need.

6 Summary

In this paper, we presented a prototype of an ORDBMS database engine that is implemented in Java. The design is a 3-layer architecture composed of the *Database*, *View*, and *Query-Command* modules. The database engine is totally self-contained, relying on no other tools except the JDK. An SQL-like query language is formally defined and the handling of the data in the database tables is at the physical storage level. The *Database* module handles the schema and access methods of the data, as well as user-defined types and methods. The *View* module is responsible for the analysis and evaluation of expressions/fields/attributes/methods in the user's query. The *Query-Command* module parses and processes user's queries.

The database engine is for object-oriented databases and hence supports objects (chunk, structured, set, ref, etc.) and methods as attributes in a table. It also supports inheritance and polymorphism. The prototype contains over 200 Java classes and interfaces with over 1,000 methods. A media database is used to test the ORDBMS and it works as expected.

We did not discuss much in detail about the implementation of polymorphism in method calls, which is basically a chain of references with OIDs associated with the objects along the chain.

We are currently study some other aspects of ORDBMS, including object indexing and optimization of query execution.

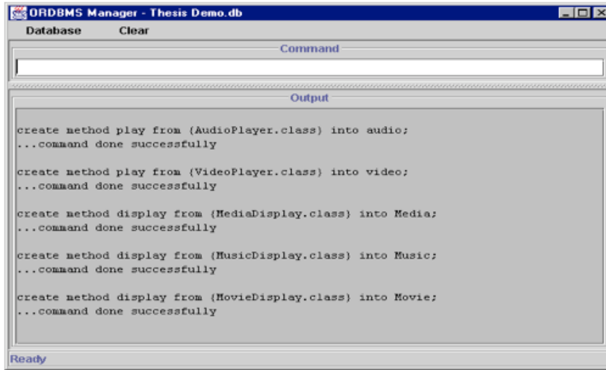

Fig. 11. Methods are associated with data types and tables. In this screen, method `play()` is created for `audio` and `video` data types that were created before (not shown), and method `display()` is created for `Media`, `Music`, and `Movie` table.

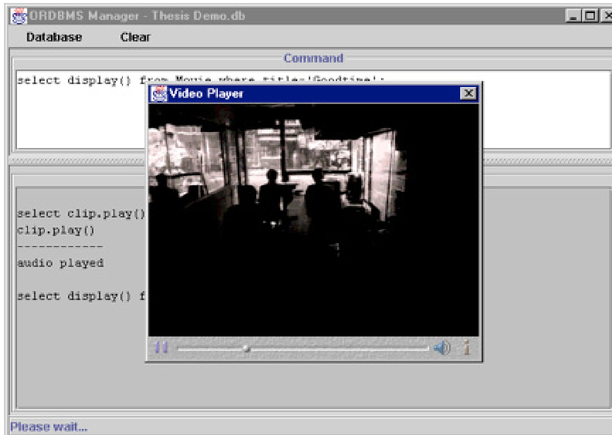

Fig. 12. The method `play()` is called from the field `clip` of `Music` table. And then, the method `display()` is also called on `Movie`. The results are both having music and movie displayed on the screen.

Acknowledgments. "This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the CITRC (Convergence Information Technology Research Center) support program (NIPA-2013-H0401-13-2008) supervised by the NIPA (National IT Industry Promotion Agency)".

"This research was also supported by the International Research & Development Program of the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning(Grant number: K 2012057499)".

References

1. Corchado, J.M., De Paz, J.F., Rodríguez, S., Bajo, J.: Model of experts for decision support in the diagnosis of leukemia patients. Artificial Intelligence in Medicine 46(3), 179–200 (2009)
2. De Paz, J.F., Bajo, J., López, V.F., Corchado, J.M.: Biomedic Organizations: An intelligent dynamicarchitecture for KDD. Information Sciences 224, 49–61 (2013)
3. Rodríguez, S., de Paz, Y., Bajo, J., Corchado, J.M.: Social-based planning model for multiagent systems. Expert Systems with Applications 38(10), 13005–13023 (2011)
4. Bajo, J., De Paz, J.F., Rodríguez, S., González, A.: Multi-agent system to monitor oceanic environments. Integrated Computer-Aided Engineering 17(2), 131–144 (2010)
5. De Paz, J.F., Rodríguez, S., Bajo, J., Corchado, J.M.: Mathematical model for dynamic case-based planning. International Journal of Computer Mathematics 86(10-11), 1719–1730 (2009)
6. Corchado, J.M., Bajo, J., De Paz, J.F., Rodríguez, S.: An execution time neural-CBR guidance assistant. Neurocomputing 72(13), 2743–2753 (2009)
7. Závodská, A., ŠRamová, V., Aho, A.M.: Knowledge in Value Creation Process for Increasing Competitive Advantage. Advances in Distributed Computing and Artificial Intelligence Journal 1(3), 35–47 (2012)
8. Satoh, I.: Bio-inspired Self-Adaptive Agents in Distributed Systems. Advances in Distributed. Computing and Artificial Intelligence Journal 1(2), 49–56 (2012)
9. Agüero, J., Rebollo, M., Carrascosa, C., Julián, V.: MDD-Approach for developing Pervasive Systems based on Service-Oriented Multi-Agent Systems. Advances in Distributed Computing and Artificial Intelligence Journal 1(6), 55–64 (2013)
10. Khoshafian, S., Dasananda, S., Minassian, N., Ketabchi, M.: The Jasmine Object Database: Multimedia Applications for the Web. Computer Associates International (1998)
11. Kim, W.: Introduction to Object-Oriented Databases. MIT Press, Cambridge (1990)
12. Larson, J.A.: Database directions: from relational to distributed, multimedia, and object-oriented database systems. Prentice Hall, Upper Saddle River (1995)
13. Lejter, M., Meyers, S., Peiss, S.P.: Support for maintaining object-oriented programs. IEEE Transactions on Software Engineering 18, 1045–1052 (1992)
14. Saracco, C.M.: Universal database management: a guide to object/relational technology. Morgan Kaufmann Publishers, San Francisco (1998)
15. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 3rd edn. McGraw Hill (1997)
16. Tesler, L.G.: Object-oriented approach. Communication of the ACM 34(8), 13–14 (1991)