EyeDB Overview

Contents

Overview

This document presents a quick overview of EYEDB. All the topics introduced here are developped in the other documents.

1   Introduction

The key features of the EYEDB OODBMS are:

- **standard OODBMS features**: persistent typed data management; client/server model; transactional services; recovery system; expressive object model; inheritance; integrity constraints; methods; triggers; query language; application programming interfaces . . .

- **language orientation**: a definition language based on the ODMG Object Definition Language (ODL); a query language based on the ODMG Object Query Language (OQL); several manipulation language bindings (at least C++ and Java),

- **genericity and orthogonality of the object model**: inspired by the SmallTalk, LOOPS, Java and ObjVlisp object models (i.e. every class derives from the class `object` and can be manipulated as an object); type polymorphism; binary relationships; literal and object types; transient and persistent objects; method and trigger overloading; template-based collections such as set, bag and array; multi-dimensional and variable size dimensional arrays,

- **support for large databases**: databases up to several Tb (tera-bytes),

- **efficiency**: database objects must be directly mapped within the virtual memory space; object memory copy must be reduced to the minimum; clever caching policies must be implemented,

- **scalability**: programs must be able to deal with hundred of millions of objects without loss of performance.

We describe below how EYEDB meets these requirements. Section 2 introduces the storage manager subsystem. In section 3, the object model is exposed. Sections 4 and 5 describe the EYEDB object definition language and query language. Sections 6 and 7 deal with the C++ and Java bindings.

2   The Architecture

EYEDB is based on a client/server architecture as shown in Figure 1.

The EYEDB server is composed of:

- the server protocol layer based on Remote Procedure Call (RPC),

- the object model implementation,

- the OQL engine,

- the storage manager subsystem.

A client is composed of:

- the user application code,

- the C++ (resp. Java) API implementing the C++ (resp. Java) binding,

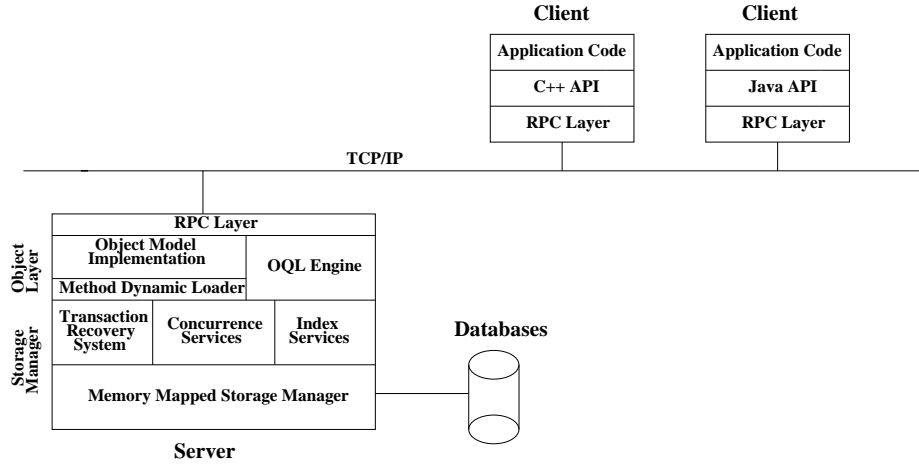- the client protocol layer based on RPC.

Figure 1: The EYEDB Architecture

3   The Storage Manager Subsystem

EYEDB is based on a client/server architecture. The server kernel is the storage manager subsystem providing the following

main services:

- persistent raw data management,
- transactional services,
- recovery system,
- B-tree and hash indexes,
- multi-volume database management.

The storage manager can be used independently from EYEDB.

3.1   Raw Data Management

The central concept of the storage manager is the raw object. A raw object is a piece of persistent raw data tied to an object identifier named *oid* .
An *oid* identifies a raw object in a unique way within a set of databases. It is generated by the storage manager at raw object creation.
An *oid* is composed of three fields: the storage index, the database identifier and a random generated magic number.
The first field identifies the object physical location within a database volume. The second one identifies a database and the last one ensures more security in the object identification process.

The storage manager is responsible for the management of raw objects:

- raw object creation (*storage allocation, oid allocation, object storage*),

- raw object update (*contents modification*),

- raw object reading,

- raw object deleting (*oid deallocation, storage deallocation*),

- raw object resizing (*storage reallocation, object moving*),

- raw object locking and unlocking (*share locking, exclusive locking, private locking*),

- raw object access control

3.2   Memory Mapped Architecture

The storage manager is based on a memory mapped architecture. Database volumes are mapped within the server virtual memory space.
Due to some 32-bit system limitations, the databases greater than 2Gb cannot be mapped as a whole.

The storage manager implements a segment-based mapping algorithm: when reading an object, the storage manager checks if the corresponding storage piece is mapped within its virtual memory space. If it is not mapped, it maps a large

segment of data around the raw object, eventually extending a neighboring segment.

If the total mapped size is more than the system maximum, it unmaps the less recent used mapped segment. On 64-bit system, this algorithm is not needed as databases up to several Tb (i.e. tera-bytes) can be mapped at whole within the virtual memory space.

Currently, the storage manager can deal with databases up to one Tb.

3.3 Transactional Services

The storage manager provides standard transaction services which guarantees atomicity, consistency, isolation and integrity within a database.

Its transaction unit is based on a two-phase locking protocol. The protocol requires that each transaction issues lock and unlock requests in two phases:

- growing phase: a transaction may obtain locks but may not release any lock.

- shrinking phase: a transaction may release locks but may not obtain any new lock.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase and no more lock requests may be issued.

The storage manager provides different transaction locking modes: read and write shared, read shared and write exclusive, read and write exclusive or database exclusive.

It provides immediate deadlock detection.

3.4 The Recovery System

The storage manager provides a simple but efficient recovery system against failures:

- client failure: the transaction is automatically aborted by the server.

- server failure or operating system failure: the current transactions will be automatically aborted on the next database opening.

- the disk failure recovery is not supported: this is a deliberate choice of simplicity since storage consistency can rely on the RAID technology or transactional file systems now available on modern operating systems.

3.5 B-Tree and Hash Indexes

The storage manager provides support for B-Tree and Hash indexes.

The B-Tree index provides fixed size raw data indexation, efficient exact match query and range query.

The Hash index provides variable size raw data indexation and efficient exact match query. The hash key function can be provided by the client.

3.6 Multi-Volume Database Management

The database storage unit is the volume files. A database can contains up to 512 volumes each one up to 2Gb on a 32-bit file system interface, or up to several tera-bytes on a 64-bit file system interface. The storage manager provides facilities to add, move, resize and reorganize database volumes.

4 The Object Model

The EYEDB object model is inspired by the SmallTalk, LOOPS, ObjVlisp, Java and ODMG models.

The main three class abstractions are the class `object` which is the root class, the class `class` and the class `instance` as shown in Figure 2.

Generally speaking, the instantiation of a class `X` gives an instance of the class `X`.

An instance cannot be instantiated except the instances of the class `class` or its subclasses: the instantiation of an instance of the class `class` is an instance of the class `instance` (i.e. an instance of the class `instance`).

If *new()* denotes the instantiation method:

```
struct_class Person =
    struct_class->new(name = "Person", ...)
```
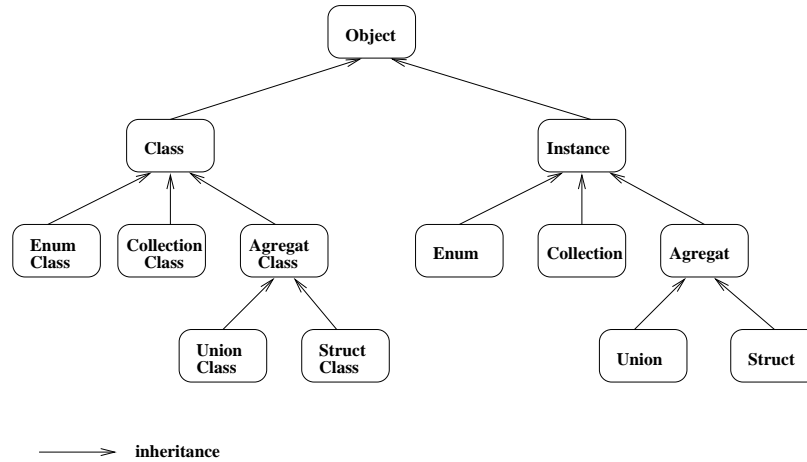
Figure 2: Partial Native Object Model

Person is an instance of the class `struct_class` that can be instantiated:

```
struct john =
    Person->new(name = "john", age = 32)
```

john is an instance of the class `struct` that cannot be instantiated

```
struct_class Employee =
    struct_class->new(name = "Employee",
    parent = Person, ...)
```

```
struct henry =
    Employee->new(name = "henry",
    salary = 10000)
```

Figure 3 shows this instantiation mechanisms.

Note that as the class `class` derives from the class `object`, an instance of the class `class` can be manipulated like any instance of the class `object`.

The native EYEDB object model is composed of 76 classes such as the class `collection`, the class `method`, the class `constraint`, the class `index`, the class `image` and so on.
EYEDB object model supports all standard built-in types: `16-bit`, `32-bit` and `64-bit integer`, `character`, `string`, `64-bit float`.

An instance can be *transient* or *persistent*:

- an instance is *transient* if its lifetime does not exceed the lifetime of the unit of execution in which it is manipulated.

- otherwise the instance is *persistent*.

A *persistent* instance can be *object* or *literal*:

- an *object persistent* instance has an unique identifier (i.e. an *oid*)

- a *literal persistent* instance has no identifier.

4.1   Class Structure

A class is composed of a name, a parent class (except for the class `object` which is the root class), a set of attributes, a set of methods and a set of triggers:

- an attribute is composed of a type, an optionnal array modifier and is *literal* or *object*. For instance, using the EYEDB ODL language:

    ```
    attribute int32 age
    ```

Object

Class   Instance

Struct   Struct
Class

Person   john

mary

instantiation
inheritance   Employee   henry
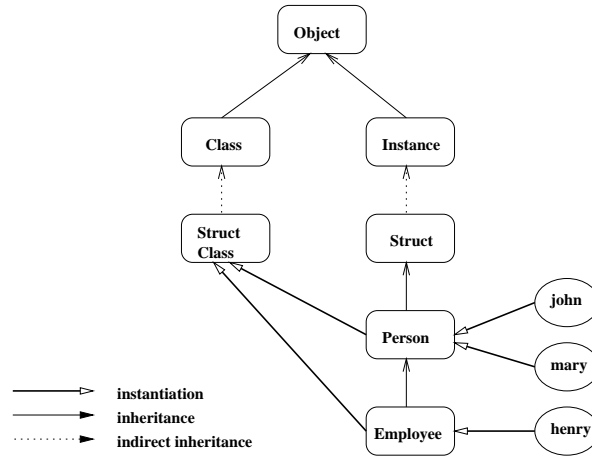indirect inheritance

Figure 3: Applicative Object Model Example

is a *literal* attribute of type `int32` with no array modifier, while the following attribute:

```
attribute Person *children[10]
```

is a fixed-size array of *object* of type `Person`.

- a method is a unit of execution tied to a class.
  A method can be either a class method or an instance method.

- a trigger is a unit of execution tied to a class. Triggers are applied to instances of this class on a given event.
  For example, a trigger `update_before` tied to the class `X` means that before the update of any instance of the class `X`, the trigger will be called.
  A method or a trigger can be overloaded by the sub-classes.

  EYEDB supports the following trigger events: `create_before`, `create_after`, `update_before`, `update_after`, `load_before`, `load_after`, `remove_before`, `remove_after`.

4.2 Type Polymorphism

The two language bindings, C++ and Java, and EYEDB OQL supports type polymorphism: variables may be bound by instances of different types.
This is a direct consequence owing to the fact that any EYEDB class inherits from the class `object`,

The possibility of manipulating polymorphic objects is a major contribution of object orientation.

4.3 The Collection Type

A collection is composed of elements of the same type.
The elements can be either *literal* or *object*.

If the collection element type is the class `object`, then the collection can contain instances of any class, as all classes inherit from the class `object`.

The collection types supported by EYEDB are the `set`, the `bag` and the `array`:

- an instance of the class `set` is an unordered collection with no duplicates allowed,

- an instance of the class `bag` is an unordered collection that may contain duplicates,

- an instance of the class `array` instance is dynamically sized ordered collection.

The collection type is a major concept of the EYEDB object model.

4.4   Relationships

The EYEDB object model supports only binary relationships, i.e. relationships between two types.
A binary relationship may be one-to-one, one-to-many or many-to-many depending on the cardinality of the related types.
Relationships are not named.

EYEDB maintains the referential integrity of relationships. This means that if an object that participates in a relationship is removed, then any traversal path to that object is also removed.
EYEDB supports object-valued attribute: this kind of attribute enables one object to reference another without expectation of referencial integrity. An object-valued attribute implements a unidirectionnal relationship: in this case, EYEDB does not guarantee the referential integrity. Note that such a unidirectionnal relationship is not called a relationship.

The example introduced in the section *The Object Definition Language* illustrates the use of relationships and object-valued attributes.

4.5   Constraints

EYEDB supports all standard constraints:

- the `not null` constraint on a attribute within a class `X` means that no instances of the class `X` can have this attribute value not assigned.

- the `unique` constraint on a attribute within a class `X` means that one cannot create an instance of the class `X` which has the same

  attribute value than an existing instance in the database.

- the `cardinality` constraint on an instance of the class `collection` means that the count of this collection must follow this cardinality constraint.

5   The Object Definition Language

The EYEDB Object Definition Language (ODL) is a language based on the ODMG ODL to define the specifications of object types.
ODL is not intended to be a full programming language, it is a definition language for objet specifications.

Like ODMG ODL, EYEDB ODL defines classes (inheritance and attributes), relationships and method signatures. EYEDB ODL extends the ODMG ODL to allow the definition of attribute constraints (notnull, unique, collection cardinality), index specifications and trigger declarations. Unlike ODMG ODL, any instance of a class can be used either as a *literal* or as an *object*. EYEDB ODL also allows the user to specify whether a method is backend (i.e. server side) or frontend (i.e. client side), and whether it is a class or instance method.

Here is a simple example of an EYEDB ODL construct:

```
enum CivilState {
  Lady = 0x10,
  Sir  = 0x20,
  Miss = 0x40
};

class Address {
  attribute string street;
  attribute string<32> town;
};

class Person {
  attribute string name;
  attribute int age;
  attribute Address addr;
  attribute CivilState cstate;
  attribute Person * spouse inverse Person::spouse;
  attribute set<Car *> * cars inverse Car::owner;
  attribute Person *children[];

  instmethod void change_address(in string street,
```

```
                                      in string town,
                                      out string oldstreet,
                                      out string oldtown);

  classmethod int getPersonCount();
  index on name;
};

class Car {
  attribute string brand;
  attribute int num;
  attribute Person *owner inverse Person::cars;
};

class Employee extends Person {
  attribute long salary;
  Person *boss;
};
```

This example illustrates all the concepts that we described previously.
The class `Person` is composed of a number of attributes each of one having an interesting particularity.
The `name` attribute is a variable size character array, i.e. a string.
This attribute is *literal*, which means that it has no identifier within a database. The hint `index` means that this attribute should be indexed to provide efficient query according to the attribute value.

The `age` attribute is a simple *literal* `32-bit integer`.

The `addr` attribute is a *literal* user type attribute. As this attribute is *literal*, the type attribute, `Address`, must have been defined before, which is the case.

The next attribute `spouse` has two interesting particularities:

1. a * character follows the user type *Person*, meaning that this attribute is not a *literal* but an *object* (i.e. with an identifier). The * character means a reference to an object.

2. the hint `(invers Person::spouse` following `spouse` means that this attribute is a relationship.
   As the attribute `spouse` is not a collection and the target attribute `spouse` is not a collection, this is a one-to-one relationship.

The `cars` attribute has also several interesting particularities:

1. as a * character follows the user type, this is an *object*.

2. this attribute is a `set` whose elements are *object* of type `Car`. Note that the user type `Car` is defined after.

3. the hint `(inverse Car::owner` following `cars` means that this attribute is a relationship whose target is the `owner` attribute within the class `Car`.
   As the source attribute `cars` is a collection and the target attribute `owner` is not a collection, the relationship is a many-to-one relationship.

As indicated by the keyword `instmethod`, the method `change_address` is an instance method. Note that this keyword is optionnal as this is the default.
The method `getPersonCount` is a class method as indicated by the `classmethod` keyword.

The class `Employee` inherits from the class `Person` as indicated by the keyword `extends`. It introduces two attributes `salary`, a *literal* integer attribute and `boss`, an *object* attribute which reference an instance of the class `Person`. Note that as there is no relationship indication (i.e. `inverse` keyword), the `boss` attribute is an object-valued attribute (i.e. a unidirectionnal relationship): in this case, EYEDB does not guarantee the referential integrity.

6 The Object Query Language

EYEDB provides a query language based on the ODMG OQL.
Although EYEDB OQL is not an OML (i.e. an Object Manipulation Language), most of the common language operations can be performed (arithmetic and logical operations, string manipulation, flow control, function definition) as well as query constructs.

EYEDB OQL adds a few features from the ODMG OQL such as flow control (`if else`, `for`, `while`), function defini-
tion, an assignement operator, and regular expression operators.

For instance the following examples are EYEDB OQL legal constructs:

```
function max(x, y) {return (x > y ? x : y);};

function fib(n) {
  if (n < 2)
    return n;
  return fib(n-1) + fib(n-2);
};

for (x in list(1, 2, 3, 4))
  fib(x);

for (x := 0; x < 10; x++)
  fib(x);
```

Note that the previous code does not perform any query.

The following code perform queries:

```
select Person;           // returns the OIDs of all Person instances

select x from Person;    // idem

select Person.name =  "john"; // returns the Person whose name is "john"

select Person.name ~ "^a.*b"; // returns the instances whose name matches
                              // the regular expression

select Person.name !~~ "ja" // returns the Person whose name does
                            // not matches the regular expression in a case
                            // insensitive way.

select x from Person x where x.age > 2 && // returns Person  whose age is between
                     x.age < 10;          // 2 and 10.

select Person.name;      // returns all Person names

select x.name from Person x;  // idem

for (x in (select Person)) // for each Person
 if (x.name ~ "^j")        // whose name matches
   x.name := \"_\" +       // the regular expression
       x.name;;            // "j", adds a "_" before
                           // the name.

// set the age of the Person whose name
// is "john" to 20:
(select Person.name = "john").age := 20;
```

7   The C++ Binding

The C++ binding maps the EYEDB object model into C++ by introducing a generic API and a tool to generate a specific
C++ API from a given schema, built upon the generic API.

Each class in the EYEDB object model is implemented as a C++ class within the C++ API: there is a one-to-one
mapping between the object model and the C++ API.

7.1   Transient and Persistent Objects

There are two types of runtime objects: persistent runtime objects and transient runtime objects.
A runtime object is persistent if it is tied to a database object. Otherwise, it is transient.

By default, EYEDB does not provide an automatic synchronisation between persistent runtime objects and database objects.
When setting values on a persistent runtime object, we do not modify the tied database object. One must call the `store` method on the persistent runtime object to update the tied database object.

Note that any persistent runtime object manipulation must be done in the scope of a transaction.

To illustrate object manipulations, we introduce a simple concrete example using the schema-oriented C++ API, based on the previous ODL example construct:

```
// connecting to the EyeDB server
eyedb::Connection conn;
conn.open();

// opening database dbname
personDatabase db(dbname);
db.open(&conn, eyedb::Database::DBRW);

// beginning a transaction
db.transactionBegin();

// creating a Person
Person *p = new Person(&db);

// setting attribute values
p->setCstate(Sir);
p->setName(name);
p->setAge(age);

p->getAddr()->setStreet("voltaire");
p->getAddr()->setTown("paris");

// creating two cars
Car *car1 = new Car(&db);
car1->setBrand("renault");
car1->setNum(18374);

Car *car2 = new Car(&db);
car2->setBrand("ford");
car2->setNum(233491);

// adding the cars to the created person
p->addToCarsColl(car1);
p->addToCarsColl(car2);

// storing all in database
p->store(eyedb::RecMode::FullRecurs);

// committing the transaction
db.transactionCommit();
```

A few remarks about this code:

1. the statement `Person *p = new Person(&db)` creates a transient runtime object. This runtime object is not tied to any database object until the `store` method has been called.

2. all the selector and modifier methods such as `setName`, `getAddr`, `addToCarsColl` have been generated by the EYEDB ODL compiler from the previous ODL construct.

3. the `eyedb::RecMode::FullRecurs` argument to the `store` method allows the user for storing each object related the calling instance: so the runtime object `car1` and `car2` within the `cars` collection will be automatically stored using the `store` method with this argument.

4. the call to `transactionCommit` ensures that the database changes will be kept in the database.

8   The Java Binding

The use of the Java language for an EYEDB binding has been motivated by several reasons:

1. Java is architecture independent,

2. Java is valuable for distributed network environment,

3. Java has a very rich builtin library,

4. Java is secure,

5. Java is easier to program than C++.

The Java binding is very close from the C++ binding: the class interfaces are identical, the functionalities are the same; only the language is slightly different.

The previous C++ code is here translated for the EYEDB Java API:

```
// connecting to the EyeDB server
org.eyedb.Connection conn = new org.eyedb.Connection();

// opening database dbname
person.Database db = new person.Database(dbname);
db.open(conn, org.eyedb.Database.DBRW);

// beginning a transaction
db.transactionBegin();

// creating a Person
Person p = new Person(db);

// setting attribute values
p.setCstate(CivilState.Sir);
p.setName(name);
p.setAge(age);

p.getAddr().setStreet("voltaire");
p.getAddr().setTown("paris");

// creating two cars
Car car1 = new Car(db);
car1.setBrand("renault");
car1.setNum(18374);

Car car2 = new Car(db);
car2.setBrand("ford");
car2.setNum(233491);

// adding the cars to the created person
p.addToCarsColl(car1);
p.addToCarsColl(car2);

// storing all in database
p.store(org.eyedb.RecMode::FullRecurs);

// committing the transaction
db.transactionCommit();
```

As shown in this example, the code is absolutely identical except that that some `->` in C++ are replaced by a `.` character in Java.
The only difference that does not appear in our examples is the object memory management. In the C++ example, one should release all the allocated objects; it is not necessary in Java.

9 Conclusion

This chapter provided a quick overview of the EYEDB OODBMS. The next chapter provides a more pragmatical approach of EYEDB by working through a simple example of defining an ODL schema and manipulating persistent data in OQL, C++ and Java.