TU Muenchen

Hauptseminar / WS2001/02

Database Hall of Fame

David Maier

Object-Oriented Database Theory

An Introduction

&

Indexing in OODBS

Prepared by: Ming-Ju Lee

Supervisor: Andreas Gruenhagen

Content

1 Introduction

1.1 David Maier

Dr. David Maier, born on 2 June 1953 in Eugene, Oregon, is a professor of Computer Science and Engineering at Oregon Graduate Institute of Science & Technology since 1988. His holds a B.A. in Mathematics and Computer Science from the University of Oregon (Honors College, 1974) and a Ph.D. in Electrical Engineering and Computer Science by Princeton University (1978).

He has been chairman of the program committee of the ACM International Conference on the Management of Data, and served on the committees for the ACM Symposium on Principles of Database Systems and the first conference on Object-Oriented Programming Systems, Languages and Applications. He also served as an associate editor of ACM Transactions on Database Systems. Dr. Maier has consulted with Tektronix, Inc., Servio Corporation, the Microelectronics and Computer Technology Corporation (MCC), Digital Equipment Corporation, Altair, Honeywell, Texas Instruments, IBM, Microsoft, Informix, Oracle, NCR, and Object Design, as well as several governmental agencies. He is a founding member of the Data-Intensive Systems Center (DISC), a joint project of OGI and Portland State University. He is the author of books on relational databases, logic programming and object-oriented databases, as well as papers in database theory, object-oriented technology and scientific databases. He received the Presidential Young Investigator Award from the National Science Foundation in 1984. He is also an ACM Fellow.

During his consultancy time (1983-1989) at Servio Logic Corporation, GemStone, an object-oriented database management system, was developed and entered the market as the first commercial product in 1987.

For his contributions in objects and databases, David Maier was awarded the 1997 SIGMOD Innovations Award.

1.2 Overview

There are two main parts in this paper; the next chapter gives an introduction of Object-oriented database concepts and overview of GemStone. Chapter three discusses the issues and an approach of indexing in OODBS.

2 Object-Oriented Databases

2.1 Motivation

The relational model is the basis of many commercial relational DBMS products (e.g., DB2, Informix, Oracle, Sybase) and the structured query language (SQL) is a widely accepted standard for both retrieving and updating data.

The basic relational model is simple and mainly views data as tables of rows and columns. The types of data that can be stored in a table are basic types such as integer, string, and decimal.

Relational DBMSs have been extremely successful in the market. However, the traditional RDBMSs are not suitable for applications with complex data structures or new data types for large, unstructured objects, such as CAD/CAM, Geographic information systems, multimedia databases, imaging and graphics. The RDBMSs typically do not allow users to extend the type system by adding new data types. They also only support first-normal-form relations in which the type of every column must be atomic, i.e., no sets, lists, or tables are allowed inside a column.

Due to the new needs in database systems, a number of researches for OODBMS have begun in the early 80's.

2.2 Concept & Features

While a relational database system has a clear specification given by Codd, no such specification existed for object-oriented database systems even when there were already products in the market. A consideration of the features of both object-oriented systems and database management systems has lead to a definition of an object-oriented database, which was presented at the First International Conference on Deductive, and Object-oriented Databases in the form of a manifesto in 1989. This 'manifesto' distinguishes between the mandatory, optional and open features of an object-oriented database.

The mandatory features, which must be present if the system is to be considered (in the

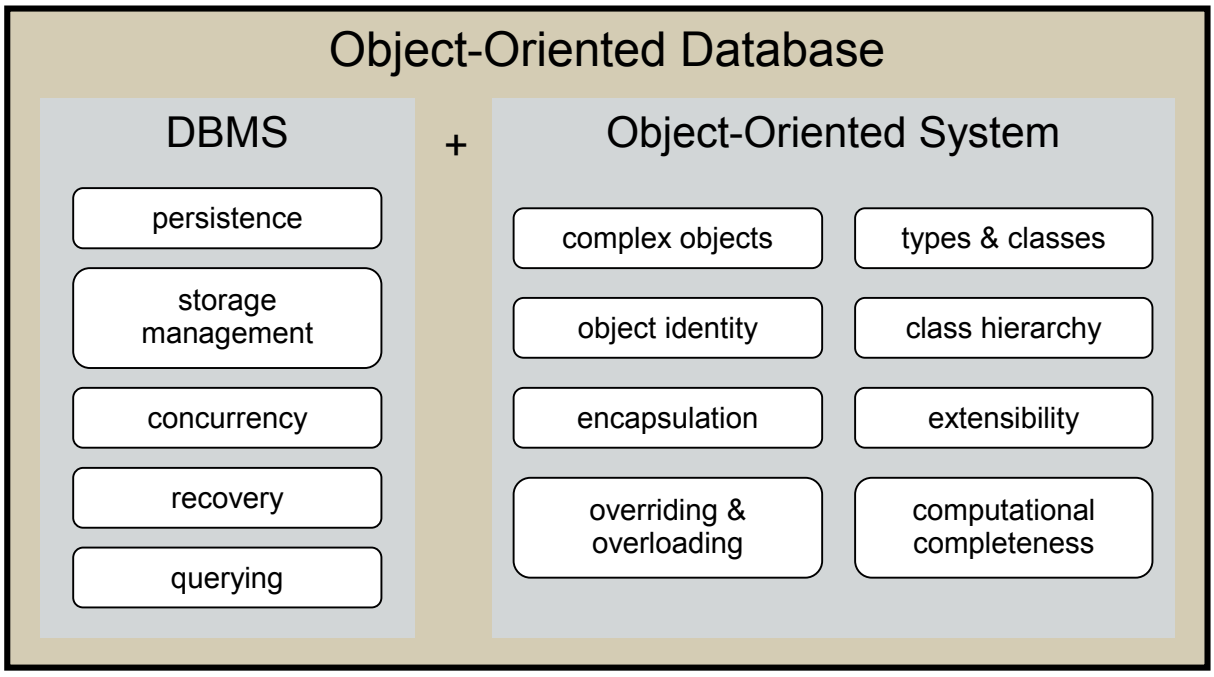


 Figure 1 OODB feathues

opinion of the manifesto authors) to be an object-oriented database, are defined in the following two paragraphs. The first part describes features of object-oriented system, as the second part features of database system.

2.2.1 Mandatory features of object-oriented systems

Support for complex objects
A *complex object* mechanism allows an object to contain attributes that can themselves be objects. In other words, the schema of an object is not in first-normal-form. Examples of attributes that can comprise a *complex object* include lists, bags, and embedded objects.
Object identity
Every instance in the database has a unique identifier (OID), which is a property of an object that distinguishes it from all other objects and remains for the lifetime of the object. In object-oriented systems, an object has an existence (identity) independent of its value.
Encapsulation
Object-oriented models enforce *encapsulation* and *information hiding*. This means, the state of objects can be manipulated and read only by invoking operations that are specified within the type definition and made visible through the **public** clause.
In an object-oriented database system encapsulation is achieved if only the operations are visible to the programmer and both the data and the implementation are hidden.
Support for types or classes
- *Type*: in an object-oriented system, summarizes the common features of a set of objects with the same characteristics. In programming languages *types* can be used at compilation time to check the correctness of programs.
- *Class*: The concept is similar to type but associated with run-time execution. The term *class* refers to a collection of all objects with the same internal structure (attributes) and methods. These objects are called instances of the *class*.
- Both of these two features can be used to group similar objects together, but it is normal for a system to support either *classes* or *types* and not both.

Class or type hierarchies
Any *subclass* or *subtype* will inherit attributes and methods from its superclass or supertype.
Overriding, Overloading and Late Binding
- *Overloading:* A class modifies an existing method, by using the same name, but with a different list, or type, of parameters.
- *Overriding*: The implementation of the operation will depend on the type of the object it is applied to.
- *Late binding*: The implementation code cannot be referenced until run-time.

Computational Completeness
SQL does not have the full power of a conventional programming language. Languages such as Pascal or C are said to be computationally complete because they can exploit the full capabilities of a computer. SQL is only relationally complete, that is, it has the full power of relational algebra. Whilst any SQL code could be rewritten as a C++ program, not all C++ programs could be rewritten in SQL.

For this reason most relational database applications involve the use of SQL embedded within a conventional programming language. The problem with this approach is that whilst SQL deals with sets of records, programming languages tend to work on a record at a time basis. This difficulty is known as the impedance mismatch. Object-oriented databases attempt to provide a seamless join between program and database and hence overcome the impedance mismatch. To make this possible the data manipulation language of an object-oriented database should be computationally complete.

2.2.2 Mandatory features of database systems

A **database** is a collection of data that is organized so that its contents can easily be accessed, managed, and updated. Thus, a database system contains the five following features:

Persistence
As in a conventional database, data must remain after the process that created it has terminated. For this purpose data has to be stored permanently on secondary storage.
Secondary Storage Management
Traditional databases employ techniques, which manage secondary storage in order to improve the performance of the system. These are usually invisible to the user of the system.
Concurrency
The system should provide a concurrency mechanism, which is similar to the concurrency mechanisms in conventional databases.
Recovery
The system should provide a recovery mechanism similar to recovery mechanisms in conventional databases.
Ad hoc query facility
The database should provide a high-level, efficient, application independent query facility. This needs not necessarily be a query language but could instead, be some type of graphical interface.

The above criteria are perhaps the most complete attempt so far to define the features of an object-oriented database in 1989. Further attempts to define an OODBS standard were made variables of researchers. One of them is a group called Object Data Management Group (ODMG). They have worked on an OODBS standard for the industry. The recent release is ODMG-2 in1997.

2.3 Making OOPL a Database

Basically, an OODBMS is an object database that provides DBMS capabilities to objects that have been created using an object-oriented programming language (OOPL). The basic principle is to add persistence to objects and to make objects persistent. Consequently application programmers who use OODBMSs typically write programs in a native OOPL such as Java, C++ or Smalltalk, and the language has some kind of Persistent class, Database class, Database Interface, or Database API that provides DBMS functionality as, effectively, an extension of the OOPL.

Object-oriented DBMSs, however, go much beyond simply adding persistence to any one object-oriented programming language. This is because, historically, many object-oriented DBMSs were built to serve the market for computer-aided design/computer-aided manufacturing (CAD/CAM) applications in which features like fast navigational access, versions, and long transactions are extremely important. Object-oriented DBMSs, therefore, support advanced object-oriented database applications with features like support for persistent objects from more than one programming language, distribution of data, advanced transaction models, versions, schema evolution, and dynamic generation of new types.

The following subsection describes object data modeling and the persistency concept in OODB.

2.3.1 Object data modeling

An object consists of three parts: structure (attribute, and relationship to other objects like aggregation, and association), behavior (a set of operations) and characteristic of types (generalization/serialization). An object is similar to an entity in ER model; therefore we begin with an example to demonstrate the structure and relationship.

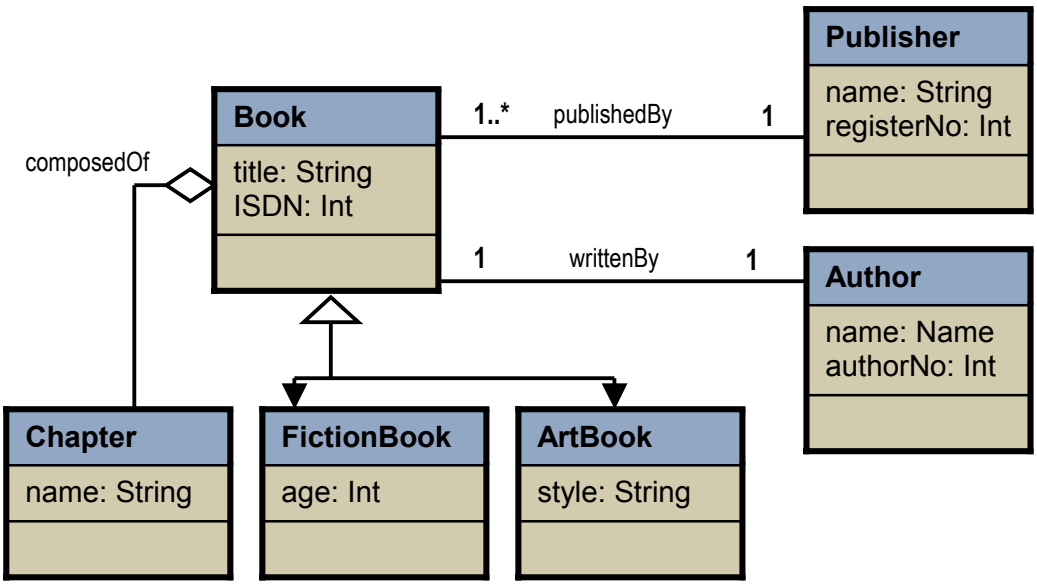


Figure 2 Book example

The structure of an object `Book` is defined as following:

```
class Book {
      title: String;
      ISDN: Int;
      publishedBy: Publisher inverse publish;
      writtenBy: Author inverse write;
      chapterSet: Set<Chapter>;
}

class Author {
      name: String;
      authorNo: Int;
      write: Book inverse writtenBy;
}
```

Attributes are like the fields in a relational model. However in the `Book` example we have, for attributes `publishedBy` and `writtenBy`, complex types `Publisher` and `Author`, which are also objects. Attributes with complex objects, in RDNS, are usually other tables linked by keys to the employee table.

Relationships: `publish` and `writtenBy` are associations with I:N and 1:1 relationship; composed_of is an aggregation (a Book is composed of chapters). The 1:N relationship is usually realized as attributes through complex types and at the behavioral level. For example,

```
class Publisher {
      …
      publish: Set<Book> inverse publishedBy;
      …
Method insert(Book book){
      publish.add(book);
}
```

Generalization/Serialization is the `is_a` relationship, which is supported in OODB through class hierarchy. An `ArtBook` is a `Book`, therefore the `ArtBook` class is a subclass of `Book` class. A subclass inherits all the attribute and method of its superclass.

```
class ArtBook extends Book {
        style: String;
}
```

Message: means by which objects communicate, and it is a request from one object to another to execute one of its methods. For example:

```
Publisher_object.insert ("Rose", 123,…)
```

i.e. request to execute the `insert` method on a `Publisher` object **)**

Method: defines the behavior of an object. Methods can be used

– to change state by modifying its attribute values

– to query the value of selected attributes

The method that responds to the message example is the method `insert` defined in the `Publisher` class.

2.3.2 Persistence of objects

Persistence, as mentioned before, means that certain program components "survive" the termination of the program. Thus these components have to be stored permanently on secondary storage.

Typically, persistence or non-persistence is specified at object creation time. There are two possible ways to make an object persistent:

(1) explicitly call built-in function ***persistence*** – certain objects are persistent

(2) automatically make object of persistent types persistent – all objects are persistent

There are several object-oriented DBMSs in the market (e.g., Gemstone, Objectivity/DB, ObjectStore, Ontos, O2, Itasca, Matisse). These products all support an object-oriented data model. Specifically, they allow the user to create a new class with attributes and methods, have the class inherit attributes and methods from superclasses, create instances of the class each with a unique object identifier, retrieve the instances either individually or collectively and load and run methods.

Most of these OODBs support a unified programming language and database language. That is, one language (e.g., C++ or Smalltalk) in which to do both general-purpose programming and database management.

2.4 GemStone

The GemStone data management system, developed at Servio Logic, was one of the first and simplest commercial OODBMS products. It is based on Smalltalk, with very few extensions. GemStone merges object-oriented language concepts with those of database systems. And provides an object-oriented database language called OPAL which is used for data definition, data manipulation and general computation.

2.4.1 Architecture

The GemStone system exhibits *client/server* architecture, and is distributed over two processes: the *Gem* and the *Stone* processes. The Stone process, running on the server,

delivers the data management capabilities performing disk I/O, concurrency control, recovery and authorization.

Stone uses unique object id called object-oriented pointers (OOPs) to refer to objects, and an object table to map an OOP to a physical location. The Gem process runs either on the server or on a client. It comprises compilation facilities, browsing capabilities, and user authentication.

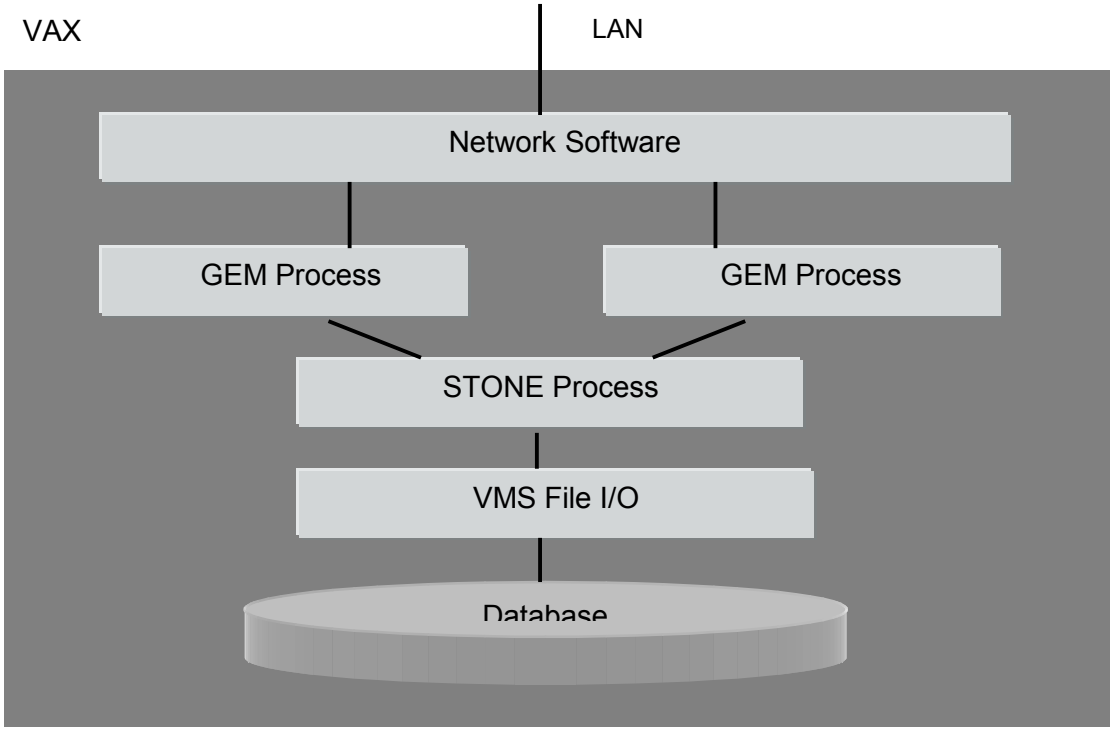


Figure 3 GemStone Architecture

2.4.2 Object model

The GemStone object model is very closely related to the Smalltalk-80 model.
The three principal concepts of the GemStone model and language are *object*, *message*, and *class*. All the objects in GemStone are made persistent.

2.4.2.1 Classes

Every GemStone object is an instance of exactly one class. Objects with the same internal structure and methods are grouped together into a class and are called instances if the class.

2.4.2.2 Objects

An object is a chunk of private memory with a public interface. Internally, most objects are divided into fields called *instance variables*. Each instance variable can hold a value, which is another object. Objects communicate with other objects by passing messages. Object is the root of all super/subclass hierarchy.

2.4.2.3 Messages

In GemStone, all actions are invoked by message passing. Messages are requests for the receiving object to change its state or return a result. The set of messages an object responds to is called *protocol* (its “public interface”). An object may be inspected or changed only through its protocol. The basic form of all message expressions is <receiver> <message>. The <receiver> part is an identifier or expression denoting an object that receives and interprets the message. The <message> part gives the selector of the message and possible arguments to the message.

2.4.2.4 Methods

The methods are the concrete implementations that can be invoked by a message sent to an instance. An object can only responds to a message if it contains a method with a selector that matches the message format. Methods are provided to query and manipulate the internal structure.

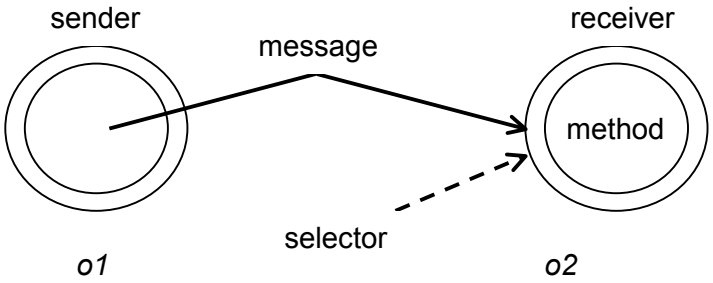


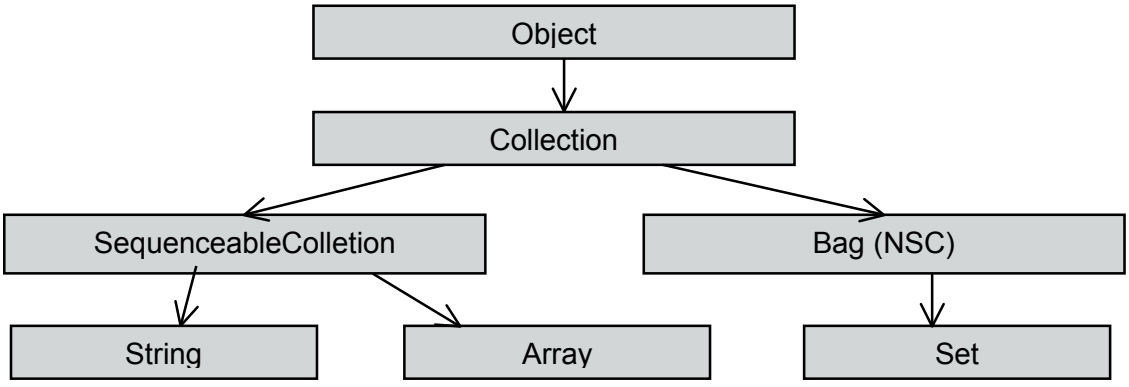
Figure 4 Message passing in GemStone

2.4.3 Collection classes

In GemStone, a class defines the structure of its instances, but rarely keeps track of all the instances. Instead, collection objects – Arrays, Bags, sets – can store groups of instances – not necessarily of the same type – in indexable or anonymous storage slots. GemStone provides built-in support for managing collections of objects by the pre-defined *Collection* class and its subclasses.

- Array: like String, is a subclass of Collection's subclass SequenceableColletion.
- Bags and Sets: are *non-sequenceable Collectio*ns, in which instance variables are *anonymous.* They do not maintain any order on their elements. The difference between a Bag and a Set is that the instances of Bag may contain the same object several times, whereas a Set contains an element just once – even though it might have been inserted several times

Figure 5 Class hierarchy of Collection classes



2.5 Comparisons of OODBS & RDBS

2.5.1 Correspondence between OODBS and RDBS

To have an idea about OODBS, the table shows the correspondence between object-oriented and relational database systems:

OODBS	RDBS
object	tuple
instance variable	column, attribute
class hierarchy	database scheme (is-a relation)
collection class	relation
OID	key
message	procedure call
method	procedure body

The correspondence between object-oriented and relational database systems

Note that this correspondence table is only an approximate equivalence. The properties in OODBS are actually not applicable in RDBS and vice versa.

2.5.2 Comparison

Although there are great advantages of using an OODBMS over an RDBMS, some disadvantages do exist. The following table shows the advantages and disadvantages using OODBS over RDBS.

Advantage	Disadvantage
▪ Complex objects & relations ▪ Class hierarchy ▪ No impedance mismatch ▪ No primary keys ▪ One data model ▪ High performance on certain tasks ▪ Less programming effort because of inheritance, re-use and extensibility of code	▪ Schema change (creating, updating…) is non trivial, it involves a system wide recompile. ▪ Lack of agreed upon standard ▪ Lack of universal query language ▪ Lack of Ad-Hoc query ▪ Language dependence: tied to a specific language ▪ Don't support a lot of concurrent users

Advantages and disadvantages using OODBS over RDBS

Because of the existing disadvantages of using OODBS, the approach of ORDBMS has become popular. In the future, it is likely that we will see the continued presence of OODBMS that address the needs of specialized market and the continued prominence of ORDBMSs that address the needs of traditional commercial markets.

The following chapter specifies the indexing design issues and its implementation in GemStone.

3 Indexing in OODBS

3.1 The basics of indexing

Indexes are essential components in database systems to speed up the evaluation of queries. To evaluate a query without an index structure, the system needs to check through the whole file to look for the desired tuple. In RBDS, indexes are especially useful when the user wishes to select a small subset of a relation's tuples based on the value of a specific attribute. In this case, the system looks up the desired attribute value in the index (stored in B-trees, or hash tables) and then retrieve the page(s) that contains the desired tuples. Using index for searching influences the performance of producing the result but not the result itself.

Indexing in OODBS is a lot more complicated than in RBDS. One difference between objects and relational tuples is that objects are not flat. Therefore one should be able to index on instance variables that are nested several levels deep in an object to be indexed.

Indexing for OODBS is first proposed for the GemStone data model. It is a generalization of an indexing technique for path expressions.

3.2 Indexing in an OODBMS

The basic need for complex structure is to efficiently select from a collection whose members meeting a selection criterion. All the objects that either contain given object, or contain an object equal to a given object have to be found.

3.2.1 Design consideration

Because of the nested and hierarchical structure of objects, it is more complicated to apply indexing on OOBDS. Several questions have to be answered to proceed the design. In this section, some essential issues will be discussed due to the features of objects; in next section, we will see how GemStone deals with these issues and how it implements indexing in its system.

3.2.1.1 Index on classes

Authorization problems occur if indexing on classes. For example, a user may have access to a `Student` object but is prohibited to the instance variable `courseHistory`. Allowing a user to build an index on `Students` could allow him to access some unauthorized information. On the other hand, if a user is prohibited to access one or some of the instances of a class, how should indexes be built in this class? For example, a professor may have access to `Students` that attend his lectures, but not other `Students`. To authorize access to certain student objects is complicated if indexing is applied on the `Student` class.

An alternative is to apply index on collections, and only add desired members to a collection; but then each object must be able to reference a number of indexes to support update, as an object may be contained in several collections

3.2.1.2 Indexing over type hierarchy

The authorization issue is also raised here when all objects of an indexed object's subclasses are also indexed. The evaluation of a query over superclass objects will retrieve also objects of its subclass. For example, the `Manager` class is a subclass of the `Employee` class. By applying index on `Employee` including `Manager`, a user who is prohibited to access the `Manager` instances can get the attribute of a manager through querying on the `Employee`.

However, if indexing on superclass and its subclasses individually, the evaluation of a query over the class hierarchy involves a lookup in several index structures and a union of the results.

3.2.1.3 Uni-directional or bi-directional index

Uni-directional index is a one-way reference from one object to another, as bi-directional index does two-way links. Two-way links have the advantage of supporting both forward and backward queries, whereas one-way link supports only one of them. Two-way link is however problematic, as an object may be the value of an instance variable in several objects. For example, the same `Publisher` instance can fill the `publishedBy` variable of many `Book` objects.
Here is an example of forward and back ward queries: `book.price` is a path,

1. `Find Books whose price are less than 100` (backward query)
2. `Find the price of the Book` *`id5`* (forward query)

3.2.2 Indexing implementation – in GemStone

In Gemstone, indexes are attached to NSCs (and only to NSCs), and only when proper typing exists for the path being indexed. Proper typing means that the variables of the last element in the path hold comparable values.

This is a middle ground solution for the problem whether to index on classes or on collection. That is to maintain a single index (per instance variable) per class, but only link members of selected collections to that index. This way, there can be different collections for differently authorized groups; also if a user is prohibited to some attribute of a class, the index on these paths that link to these attributes can't be attached to this collection.

The use of collection also solves the authorization problem of indexing on class hierarchy. Objects of subclasses can be included in their superclass collection depending on who is authorized to that collection.

As for the question of uni-directional or bi-directional index, GemStone chose the simpler way, and only support the backward query.

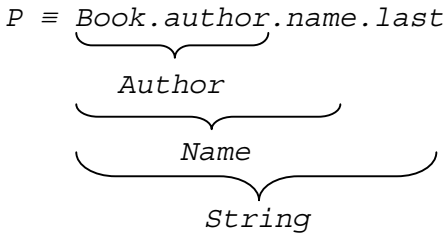
3.2.2.1 Path expression

To apply index on links, the path expression needs to be defined. A path expression (or simply a path) is a variable name followed by a sequence of zero or more instance variable names called links. The variable name appearing in a path is called the **path prefix**; the sequence of links, the **path suffix**. The value of a path expression $A.L_1.L_2. \ldots .L_n$ is defined as follows:

1. If n=0, then the value of the path expression is the value of A.
2. If n>0, then if the value of $A.L_1.L_2. \ldots .L_{n-1}$ is `nil` or `undefined`, the value of the path expression is `undefined`. Otherwise, the path expression's value is that of instance variable L_n in the value of $A.L_1.L_2. \ldots .L_n$.

A path suffix S is defined with respect to a path prefix P if the value of *P.S* is defined.

This definition distinguishes `nil` and `undefined` of the value of a path. If the value of $A.L_1.L_2. \ldots .L_{n-1}$ is defined, and the value of L_n is `nil`, then the path $A.L_1.L_2. \ldots .L_n$ leads to a value `nil`. Whereas in the case of L_n being `undefined`, the path cannot be fully traversed. Following is a valid path expression example:



3.2.2.2 Index kinds

There are two kinds of indexes supported: identity and equality indexes.

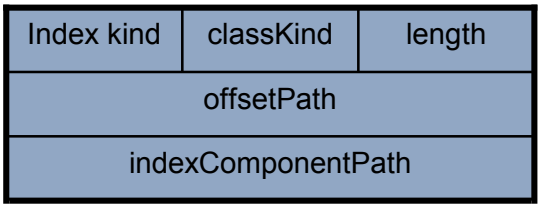
- Identity index: identity indexes support only the search operators == (identical to) and ~~ (not identical to). Since the identity of an object is independent of its class, the class kind of the final link of a path (only the final link) may be unknown.
- Equality index: equality indexes support the search operators =, ~=, <, <=, > and >=. Paths for equality indexes must lead to a `Boolean`, `Character`, `DateTime`, `Float`, `Fraction`, `Integer`, `Number`, `String` or subclasses thereof.

For `Boolean`, `Character` and `SmallInteger` as class-kinds, there is no distinction between equality and identity indexes, as the order of OOPs is the same as the order of values for these classes.

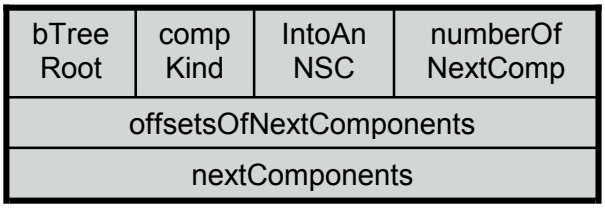
3.2.2.3 Index on paths

Indexes on paths are implemented by a sequence of index components, one for each link in the path suffix. Every NSC object has a named instance variable, NSCDict. If there is no index into an NSC, then the value of NSCDict is nil; otherwise, the value of NSCDict is the OOP of an index dictionary. An index dictionary contains the OOPs of one or more dictionary entries.

- **Dictionary entry:**
  - **Index kind:** identity or equality index
  - **Class kind** is only significant for equality indexes, and stores the class-kind of the indexed path
  - **Length** stores the length the path suffix.
  - **OffsetPath** contains an offset representation of the path suffix.
  - **IndexComponentPath** contains an OOP of the index component for each instance variable in the path suffix.



Dictionary Entry



Index Component

- **Index component**: All index components are implemented using B+-trees.
  - **BTreeRoot** contains the OOP of the root of the B+-tree of the component.
  - **CompKind** defines the ordering of keys in the component's B-tree. For all the components but the last component, the ordering is defined on the OOPs of key values. For the last component of an identity index, the ordering is also on the OOPs of key values. For the last component of an equality index, the ordering of key values is determined by the class-kind of the indexed path.

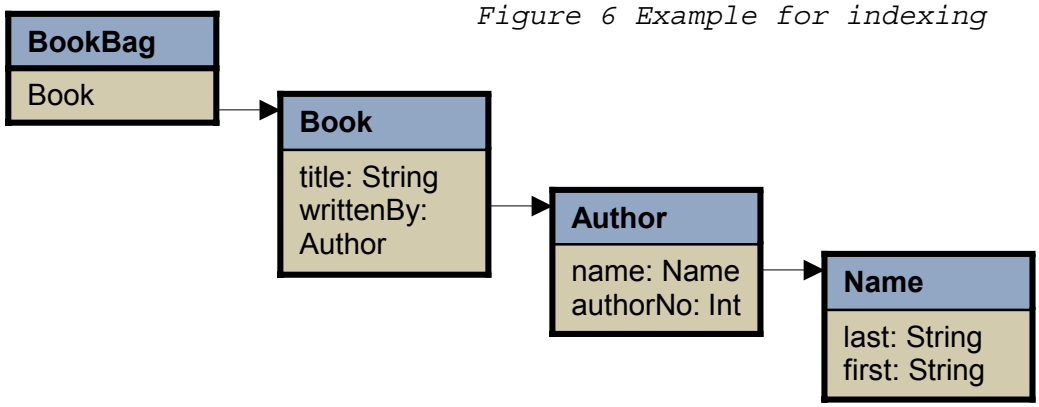
- **IntoAnNSC**: “true” for the first component of the path, which is indexed directly into an NSC.
- **OffsetsOfNextComponent**: store the offset for, and OOP of, the index component for the next link in each indexed path that shares the component.
- **NextComponent**: Parallel array to the offsetsOfNextComponents. We shall refer to the elements of the next-component.

3.2.3 Index maintenance

Every object in GemStone that participates in an index is tagged with a **dependency list.** The object dependency list contains a pair of values consisting of the OOP of the index component and the instance variable name for the component (actually the offset of the instance variable within the object).

To demonstrate index maintenance, we use the `Book` example again. `BookBag` is an `NSC` collection class, which contains `Book` object. Figure 3 shows its structure.

Figure 6 Example for indexing



Class BookBag: Bag Of BookClass;	Class Book: title: String; writtenBy: Author; …. End;	Class Author: name: Name authorNo: Int …. End;	Class Name: first, last: String; …. End;

3.2.3.1 Creation

Figure 7 shows the dictionary structure for a `BookBag` object with no extant indexes after an equality index on `title` has been created.

The first index component’s B-tree will have an entry for every element of the indexed NSC other than nil and will contain exactly one entry for each unique (by identity) non-nil `author` value of an element of the NSC.

Figure 8 shows the dictionary structures after an identity index on `author.authorNo` has been added. The first index component’s B-tree will have an entry for every element of the indexed NSC other than nil. The second component will contain exactly one entry for each unique (by identity) non-nil `author` value of an element of the NSC. Figure 9 shows an identity index on `author.name.last`.

```
BookBag createEqualityIndexOn: 'title'
```

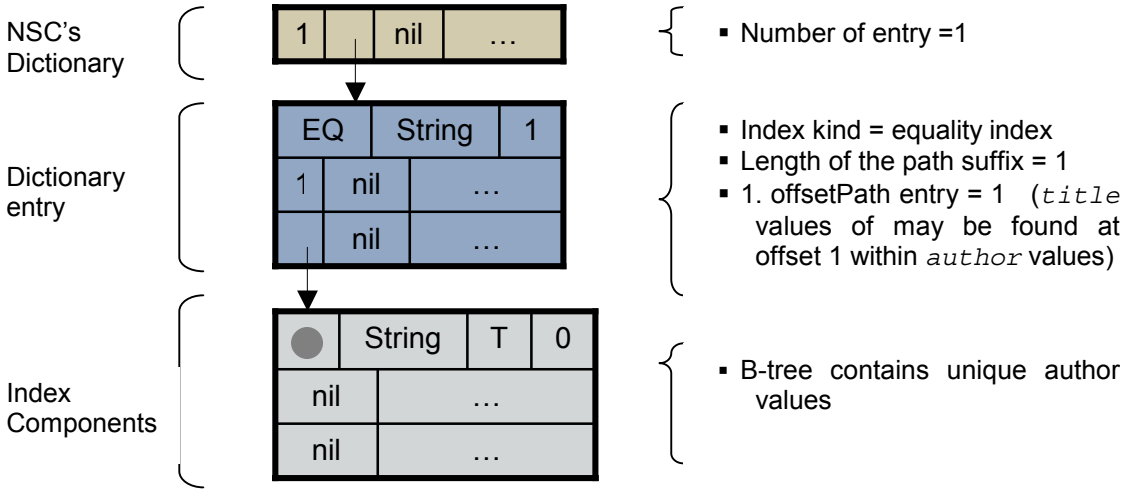


Figure 7

Both of these indexes share the component that indexes from `author` values to elements of the NSCs. The creation of the index on `author.name.last` does not require updating the B-tree of this component. This component now has three next components.

```
BookBag createIdentityIndexOn: 'author.authorNo'
```

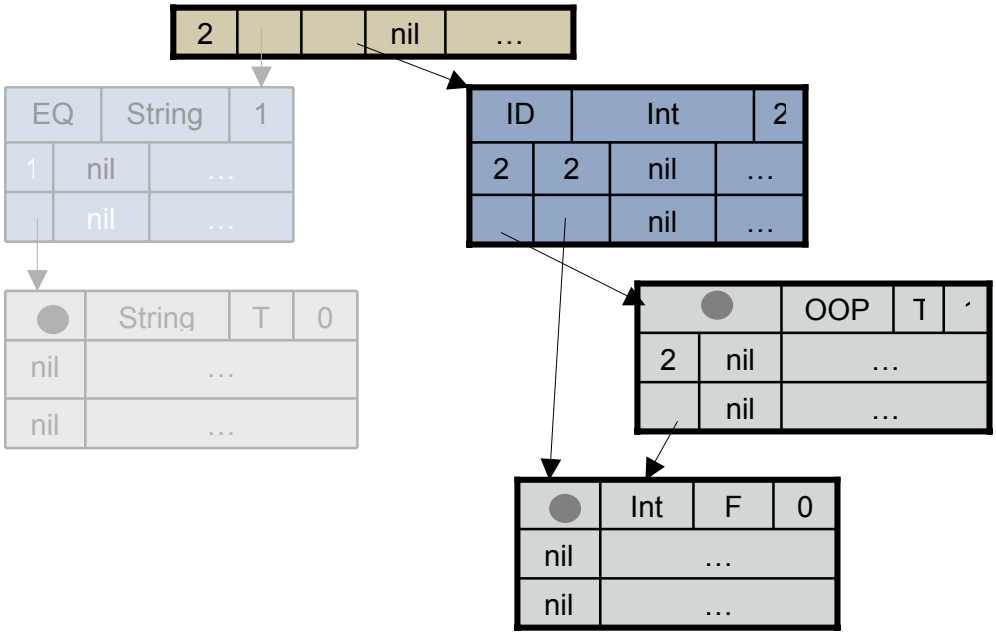


Figure 8

```
BookBag createEqualityIndexOn: 'author.name.last'
```

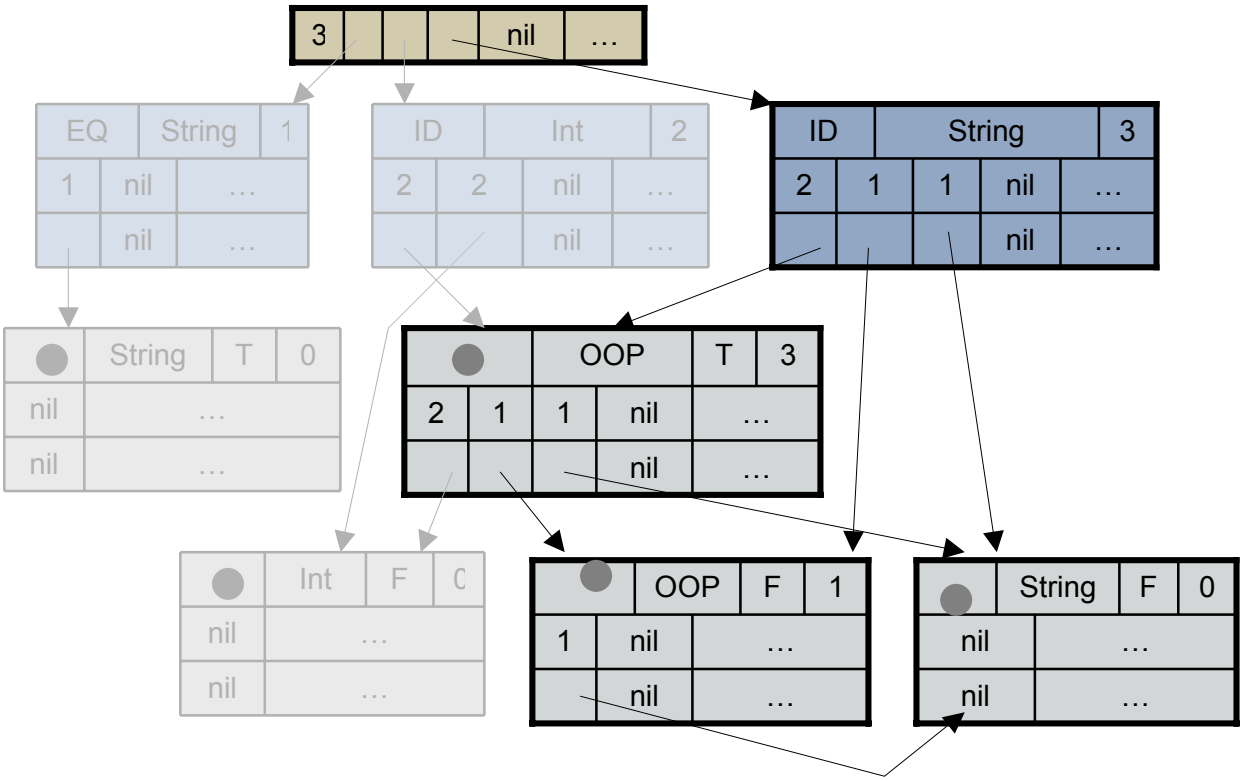


Figure 9

3.2.3.2 Removal

Consider removing the index on `author.authorNo` from the dictionary structure of the figure 10. Since the first index component is used by another indexed path, only the second index component should be deleted. In deleting the component, the entry that refers to the component must be removed from the dependency list of every object that appears as a value in the component's B-tree. Since the component is an identity component of class-kind Int, the dependency list entry that refers to the component must be removed from every object that next-component. The resulting dictionary structure is shown in Figure 7.

Remove Identity index on: author.authorNo

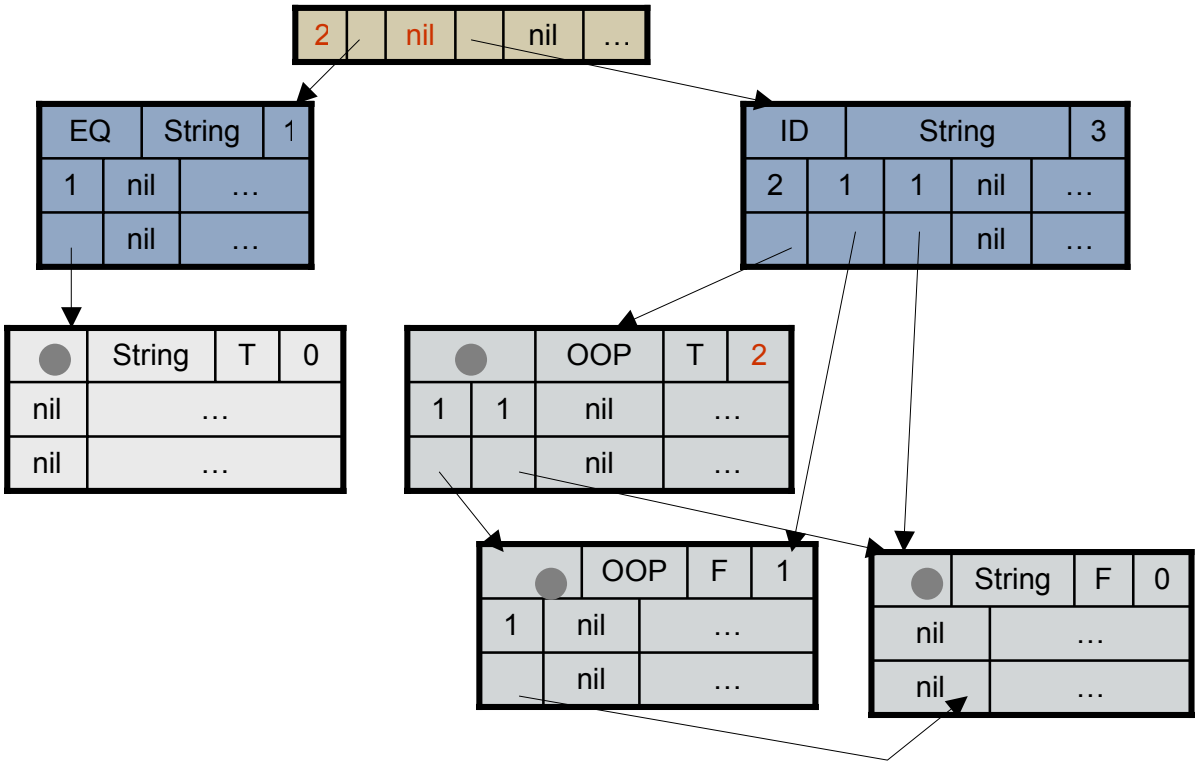


Figure 10

3.2.3.3 Object modification

When the value of an object at a given offset is modified, then a deletion followed by an insertion is made for each index component that is dependent upon the value of the object stored at that offset. When the component is not the first component of an indexComponentPath (when `intoAnNSC` is `false`), the deletion of single entry followed by the insertion of a single entry for each dependent component will do. (Note that an index component can't be a first component for one path and non-first for another.) If the dependent component references an NSC, then every occurrence of the object, old value pair in the component's B-tree must be deleted. If n occurrences are deleted then n occurrences of the object, new value pair are inserted. The propagation of these insertions and deletions is handled in the same manner as described for NSC insertion and deletion.
When a byte object with a non-nil dependency list is modified each index component on its dependency list is modified. Each entry in a dependent component's B-tree with a key value identical to the byte object is deleted from the B-tree. After the modification, each of the deleted entries is reinserted.

3.2.3.4 Indexed Lookups

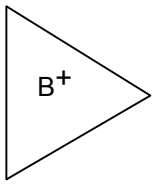
The evaluation of an indexed lookup begins with a B-tree lookup in the last index component of the indexed path's index component path. If the indexed path is of length one, then the lookup is complete. Otherwise, the following sequence is repeated *n-1* times for an indexed path of length *n*. Sort the result of the previous B-tree lookup by OOP. Using the sorted list of OOPs. Perform a lookup on the B-tree of the previous index component for the preceding link in the path.

Consider the evaluation of the term `B.author.name.last = 'Jones'`.

```
BookBag select: {B| (B.author.name.last = 'Jones')}
```

Using the B-tree from the third component of the indexed path, all those names with a `last` value of "Jones" are found. These name values are then sorted by OOP.

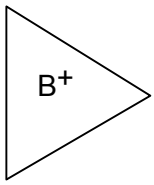
[name.last]	
OOP_name	*String*
oop3	'Becker'
oop4	'Jones'
oop2	'Jones'
oop5	'Maier'
oop1	'Wood'




```
List_OOP_name: (oop2, oop4)
```

By performing an incremental search of the B-tree of the second component, using the sorted list of `name` values as lookup keys, the elements of `Author` whose name values have a `last` value of "Jones" are found. Again the author values are stored by the OOP in a sorted list. List

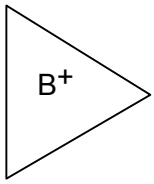
[author.name]	
OOP_author	*OOP_name*
oop12	oop1
oop14	oop2
oop15	oop3
oop11	oop4
oop13	oop5




```
List_OOP_author: (oop11, oop14)
```

Do the same as the step before, search the B-tree of the first component, using the sorted list of `author` values as lookup keys, the elements of `Book` whose author's name values have a `last` value of "Jones" are found.

[book.author]	
OOP_book	*OOP_author*
Oop39	oop11
Oop35	Oop12
Oop34	Oop13
Oop36	Oop14
Oop37	Oop15




```
Found books List_OOP_book: (oop35, oop36)
```

3.3 Summary

GemStone supports only indexing over the entire path and prohibits set-valued attributes. It can only do backward queries. Since GemStone was the first commercial product in the market, its indexing technique might not seem to be optimal, but invokes further researches and improvement in this area. One example is the work by Alfons Kemper and Guido Moerkotte. They have developed Associative Access Support (ASR) for OODBMS based on Maier's approach with more extensions and with support for set-valued attributes, also for bi-directional queries.

4 Conclusion & Outlook

The OODBMS contains extensive concepts, which makes it a lot more complicated comparing with the RDBS. This paper gives an introduction of OODBMS concepts and discusses the indexing issues in OODBMS.

The first chapter gives an overview about David Maier, who received the 1997 SIGMOD Innovations Award for his distributions in objects and databases. The indexing implementation in chapter three was one of his works. Chapter two describes the mandatory features of OODBS based on the paper `The Object-Oriented Database Manifesto`, in which David Maier has participated; it also talks about the basics of object modelling, an overview of GemStone and the comparisons of OODBs and RDBS. Chapter three specifies in one of the areas of OODBMS, indexing. It includes the issues and eventually an implementation approach of indexing in OODBMS.

The Future of the OODBMS
The market for OODBMS is growing fast, but is still dwarfed by the market for relational and object-relational databases. In 1995, the market for OODBMS drew $100 million. It is predicted to grow to $430 million by 1997 and $600 million by 2000. (In contrast, it is predicted that the market for ORDBMS will grow to $1 billion by 2000). The test will be time. In ten years, if object-oriented programming becomes the most commonly used model of programming, and if the identified limitations of the OODBMS are overcome, then we can probably anticipate that the OODBMS will be more widely used.

5 Bibliography

- Atkinson, Malcolm et al,
  The Object-Oriented Database Manifesto.
  In *Proceeding of the First International Conference on Deductive and Object-Oriented Databases*, pages 223-240, Kyoto, Japan, December 1989
- D. Maier and J. Stein,
  Development and implementation of an object-oriented DBMS.
  In *Research Directions in Object-Oriented Programming*, B. D. Shriver, P. Wegner, editors, MIT Press, 1987. Also in *Readings in Object-Oriented Database Systems*, S. Zdonik and D. Maier, editors, Morgan Kaufmann, 1990.
- D. Maier and J. Stein,
  Indexing in an object-oriented DBMS.
  In *Proceeding of the International workshop on Object-Oriented Databases*, pages 171-182, Pacific Grove, CA, September 1986
- Alfons Kemper and Guido Moerkotte,
  Object-Oriented Database Management, 1994
- David Maier's home page: www.cse.ogi.edu/~maier