SIGNATURE FILE METHODS FOR INDEXING OBJECT-ORIENTED DATABASE SYSTEMS

Wang-chien Lee and Dik L. Lee
Department of Computer and Information Science, Ohio State University
Columbus, Ohio 43210-1277

Abstract

Although object-oriented database systems offer more powerful modeling capability than relational database systems, their performance suffers from the increased complexity in the data model. Thus, efficient index mechanisms must be used to improve the performance. In this paper, two new signature methods for indexing object-oriented databases and the associated operations are described. The cost models for the analysis of storage overhead and performance of the methods are presented.

1 INTRODUCTION

In the past decade, Object-Oriented Database Systems (OODBSs) have become an important field of database research. Several experimental and commercial systems, such as GemStone [8], Orion [6] and O_2 [3]. have been built. The powerful modeling capability is a major advantage of OODBSs over relational databases. However, much work still need to be done on query processing, optimization, and indexing techniques in order to improve the performance. Most OODBSs support secondary indexes on objects to improve retrieval speed. Several indexing techniques have been proposed. For example, GemStone builds indexes along a path of object links [7]; class-hierarchy index and single-class index were investigated in the Orion project [5], although only single-class indexes are supported in the current implementation. Instead of focusing on the inheritance hierarchy of classes, other researchers explored the aggregation hierarchy of classes and proposed various indexing structures on nested attributes [1,2].

Most, if not all, indexing techniques are based on tree or network structures [1]. Key values in the index are linked with complex pointers in order to support the flexible query styles in OODBSs. The complexity results in large storage overhead and maintenance cost. In this paper, we propose a signature file approach for indexing OODBs. Signature file techniques have been extensively investigated in relational databases and text retrieval [9]. However, to the best of our knowledge, it has not been extensively researched in the context of OODBSs.

The rest of this paper is organized as follows. In Section 2, we describe the basic features of an object-oriented data model and discuss the approaches to query processing. In Section 3, the concepts and structure of signature files are discussed. We present the algorithms for query processing and signature files maintenance involving update operations in Section 4. Then a cost model is proposed in Section 5 to facilitate the analysis of overhead cost and performance of the signature mechanisms.

2 QUERIES IN OBJECT-ORIENTED DATABASES

In object-oriented database systems, an entity is represented as an object, which consists of methods and attributes. Methods are procedures and functions associated with an object defining the actions taken by the object in response to messages received. Attributes represent the state of the object. Objects having the same set of attributes and methods are grouped into the same class. A class is either a primitive class or a complex class. Objects in the respective classes are called *primitive objects* and *complex objects*. A primitive class, such as `integer` and `string`, is not further broken down into attributes or substructures. A complex class is defined by a set of attributes, which may be *primitive*, or *complex* with user-defined classes as their domains. Since a class C may have a complex attribute with domain C', a relationship can be established between C and C'. The relationship is called *aggregation relationship*. Using arrows connecting classes to represent aggregation relationship, an *aggregation hierarchy* can be constructed to show the nested structure of the classes.

Figure 1 shows the class definitions for `Person`, `Vehicle`, `Person_Name`, `Company`, and the resulting aggregation hierarchy. The class `Person` has two primitive attributes, `SSN` and `Residence`, and two complex attributes, `Owns` and

Email addresses: wang-c@cis.ohio-state.edu, dlee@cis.ohio-state.edu

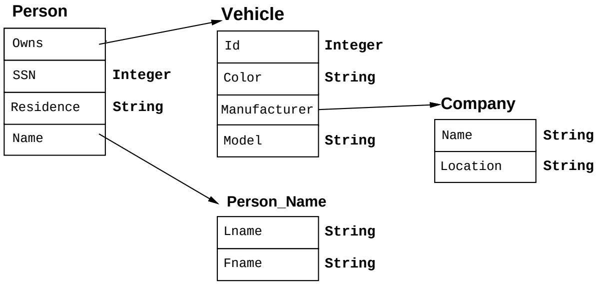


Figure 1: Aggregation hierarchy.

`Name`. The domain classes of the attributes `Owns` and `Name` are `Vehicle` and `Person_Name`, respectively. The class `Vehicle` is defined by three primitive attributes, `Id`, `Color`, and `Model`, and a complex attribute `Manufacturer`, which has domain `Company`. `Company` and `Person_Name` each consists of two primitive attributes.

Objects are nested according to the aggregation hierarchy. If an object O is referenced as an attribute of object O', O is said to be *nested* in O' and O' is referred to as the *parent* object of O. In OODBSs, each object is identified by a unique system-assigned object identifier (OID). Objects can be nested by storing the child OIDs in the complex attributes of the parent objects. Thus, child objects can be *forward referenced* from their parents through the OIDs stored in the parents. *Backward reference* from children to parents can be supported by explicitly creating a complex attribute in the child object referencing the parent (e.g., create a `owned-by` attribute in `Vehicle` with `Person` as its domain); alternatively, the system may implicitly maintains such inverse attributes. The nesting structure of objects is one of the key features which make the indexing problem of object-oriented databases different from that of relational databases.

In object-oriented databases, the search condition in a query is expressed as a boolean combination of predicates of the form <attribute operator value>. The attribute may be a nested attribute of the target class. For example, the query "retrieve all red vehicles manufactured by Fiat" can be expressed as:

```
retrieve Vehicle where Vehicle.Color = "red"
          and Vehicle.Company.Name = "Fiat"
```

The search condition against the class `Vehicle` consists of two predicates, one involving the attribute `Color` and the other involving the nested attribute `Name`.

The nested structure of an object suggests that answering a query on a class involving nested attributes requires traversal along the path from the target class to the nested attributes. There are two approaches to evaluate a query with nested attributes: top-down and bottom-up evaluations. Let's consider the above example. To answer the query in the top-down approach, the system has to retrieve all of the objects in class `Vehicle` and single out those with red color. Then, the system retrieves the company objects referenced by the red vehicles and checks the manufacturers' names. Finally, those red vehicles made by Fiat are returned. In the bottom-up approach, the objects in class `Company` are retrieved to examine if their names are Fiat. Without backward reference support, the system must keep the OIDs of the Fiat companies in a set and examine the vehicle objects in class `Vehicle` to identify those red vehicle made by the companies maintained in the set. As a result, the returned objects are red vehicles made by Fiat. With backward reference, the retrieval process is identical to the top-down approach.

The performance of the top-down and the bottom-up approaches depends on a number of factors: the sizes of the classes along the paths from the target class to the nested attributes, the conditions specified on the classes and the selectivities of the conditions, whether indexes are available for those classes, and whether backward references are supported. Assuming no indexes and backward references exist, the bottom-up approach will have to scan through all objects located in the classes on the paths. The top-down approach is in general more efficient because only forward references are needed, but the actual cost will depend on the other factors listed above.

Most of the index schemes in the literature require costly storage overhead. As a result, the number of attributes to be indexed must be kept to a minimum. Therefore, only queries involving some indexed attributes and classes benefit from these indexing mechanisms. In order to improve the overall system performance on query processing, the system has to build and maintain indexes on every attributes, which is infeasible due to the storage limitation. In this paper, we present a different secondary mechanism, namely, signature files, to improve the general performance of object-oriented database query evaluation at an affordable storage overhead.

3 SIGNATURE FILE TECHNIQUES

object	John | 123456789	
attribute signatures:		
John	001 000 110 010	
123456789	000 010 101 001	
object signature ($\vee$)	001 010 111 011	
Queries	Query Signatures	Results
1) John	000 010 101 001	← match
2) Jack	010 001 000 011	← no match
3) 111223333	001 000 111 000	← false drop

Figure 2: Signature generation and comparison.

A signature is basically an abstraction of the information stored in the (nested) attributes of an object [9]. It is a superimposed bit string generated from the attribute values of the object being represented. Figure 2 depicts the signature generation and comparison processes of an object having two primitive attributes, `name` and `SSN`. Signatures of primitive attributes are obtained by hashing their attribute values into a bit string. An object signature is formed by superimposing the signatures of its attributes. Object signatures are stored sequentially in a signature file. A query specifying certain values to be searched for is transformed into a query signature S_Q in a similar way. The query signature is then compared to every object signature in the signature file. The figure illustrates three possible outcomes of the comparison: (1) the object matches the query; that is, for every bit set in S_Q the corresponding bit in the object signature is also set (i.e. $S_Q \wedge S_i = S_Q$); (2) the object doesn't match the query (i.e. $S_Q \wedge S_i \neq S_Q$); and (3) the signature comparison indicates a match but the object in fact does not match the search criteria. The last case is called a *false drop*. In order to eliminate false drops, the object must be examined after the object signature signifies a match.

The purpose of using a signature file is to screen out most of the unqualified objects. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object accesses are prevented. Signature files have a much lower storage overhead and a simpler file structure than inverted indexes. They are particular good for multi-attribute retrieval when the attributes have equal chance of being specified in the query. Since queries in an OODBS could be very flexible, an inverted indexes would be very complex if all possible access paths are supported. However, as can be seen later, the signature file approach offers a much simpler solution at the expense of some retrieval speed.

In this section, we present two signature file techniques to support efficient object-oriented database query evaluation. Queries having a single class as target are used in this paper. However, our methods can be directly applied to queries against multiple classes in an inheritance class hierarchy.

3.1 Tree Signature Scheme

For every class C in the aggregation hierarchy, there exists a signature file S such that every object O in C has an entry $< sig, oid >$ in S, where *sig* is the signature of O and *oid* is the OID of O. The signature of an object can be recursively defined as follows.

1. the signature of an object is generated by superimposing the signatures of all of its primitive and complex attributes;
2. the signature of a primitive attribute is obtained by hashing on the attribute values; the signature of a complex attribute is the signature of the object it references.

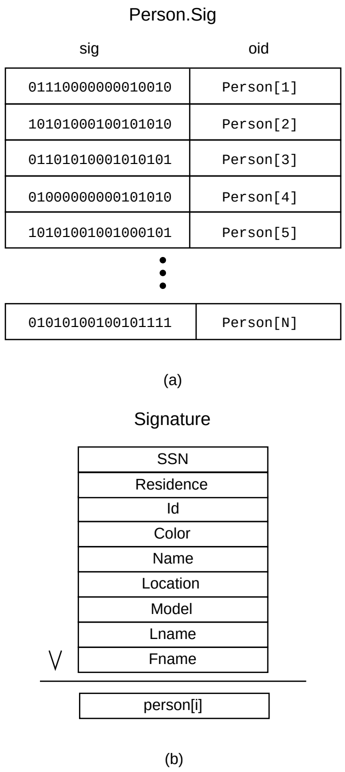


Figure 3: Tree Signature Scheme. (a) Format of a signature file; (b) creation of an object signature.

For each class in the aggregation hierarchy, a file of signatures is created for the objects in the class. The information stored in a signature file is the superimposed information of all (nested) attributes of its associated class. Figure 3 shows the format of the signature file for class `Person`. We assume that there are N objects in the class and use `Person`[i] to represent the object identifier of the ith object in `Person`. The signature file `Person.Sig` of class `Person` consists of N signatures, one for each object in the class. As shown in Fig. 3(b), the signature of an object in `Person` is the superimposition of the signatures of its primitive attributes, `SSN` and `Residence`, and all of the primitive attributes nested under `Owns` and `Name`.

3.2 Path Signature Scheme

In the tree signature scheme, a signature for an object O is created from all of the primitive values in the subtree of objects rooted at O. The filtering capability of the signature technique is weaken if the subtree is large. The path signature scheme creates signature files for a single path. The signature files created for the classes on a path will support the evaluation of queries involving any classes on the path and their attributes. Since the attributes covered in a signature is limited to those on a path, the accuracy of the information abstracted in the signature is improved. In addition to improve the filtering capability, the path signature scheme is especially useful for applications where some paths are frequently specified in queries.

Similar to the tree signature scheme, the signature of an object O in the path signature scheme is generated by superimposing the signatures of O's attributes. However, in the path signature scheme, the signatures of complex attributes located on the path and those not located on the path are generated in different ways. The signatures of complex attributes located on the path are generated by superimposing the signatures of its nested attributes, while the signatures of other complex attributes are generated by using the object identifiers stored in the attributes as the keys for hashing. In other words, the complex attributes which are not located on the path are treated as primitive attributes by taking their OIDs as primitive values. Another difference between the path and tree signature schemes is that the path signature scheme uses a list of OIDs, which includes the object identifiers of O and its nested objects on the path, in the signature files.

The formal description of the path signature scheme is as follows.

For every class C in a given path of the aggregation hierarchy, there exists a signature file S such that every object O in C has an entry $< sig, oid\_list >$ in S, where sig is the signature of O and oid_list is the object identifiers of O and its nested objects located in the path.

The signature of an object is created as follows:

1. the signature of a primitive object is generated by hashing the primitive value.

2. the signature of an object not on the path is generated by hashing on its object identifier.

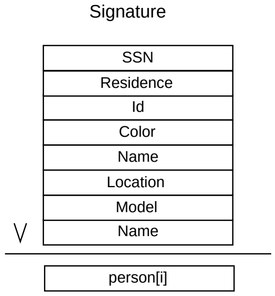
3. the signature of an object on the specified path is generated from superimposing the signatures of all of its attributes.

For a given path in the aggregation hierarchy, files of signatures are created for classes located on the path. Figure 4 shows the signature file for class `Person` located on the path `Person.Vehicle.Company`. The signature of `Person`[i] is the superimposition of the signatures for `SSN`, `Residence`, `Name`, and `Owns`. The signature of `Name` is generated by hashing its OID (`Lname` and `Fname` are not used). The signature of `Owns` is generated from all of the primitive values along the path (i.e., `Id`, `Color`, `Model`, `Name`, and `Location`).

Person.Sig

sig	oid_list		
01110000000000	Person[i]	Vehicle[100]	Company[2]
10101000100101	Person[2]	Vehicle[5]	Company[2]
01101010001010	Person[3]	Vehicle[73]	Company[10]
01000000101010	Person[4]	Vehicle[15]	Company[23]
	⋮		
01010010001001	Person[N]	Vehicle[9]	Company[16]

(a)



(b)

Figure 4: Path Signature Scheme. (a) Format of a signature file; (b) creation of an object signature.

In the signature file of class `Person`, the signature of object `Person`[i] is associated with a list of OIDs, including `Person`[i] and its nested objects. The purpose of the list is to reduce the expensive traversals between objects. The oid_list includes all nested objects an object directly or indirectly references. Therefore, when a signature matches, we can bypass the intermediate objects and directly retrieve the objects with the specified nested attributes for verification. Actually a tree of object identifiers may be used in the tree signature files. However, the maintenance of a tree of OIDs will be too complex. Since a simple list of OIDs is quite simple to maintain, it is chosen in the path signature scheme.

4 QUERY PROCESSING AND SIGNATURE MAINTENANCE

4.1 Retrieval

Tree Signature

As we mentioned before, there are two methods to evaluate a query: top-down and bottom-up approaches. The top-down approach is to retrieve all of the objects along the path from the target class to their nested attributes specified in the search condition of the query. Then, the value of the nested attribute is checked to decide if it is a desired object or not. With the signature file, the query is evaluated as follows. A query signature S_Q for the query Q is generated. S_Q is compared with every signature stored in the signature file associated with the target class. If a signature matches with S_Q, traverse the path to verify the nested attribute. In the following, the algorithm for top-down retrieval is presented.

Algorithm Top-Down Retrieval;
Given an object query Q,
retrieve a set of OIDs specified in the query.

1. Compute the query signature S_Q for the query Q.
2. For every entry (sig_i, oid_i) of the signature file associated with the target class. Compare S_Q with sig_i. If sig_i matches S_Q, then put oid_i in the returning set S.
3. For each object in S, traverse the path from the object to the nested attributes specified in Q to eliminate false-drop.

Path Signature

The structure of the path signature file benefits the top-down evaluation of the object queries more than the bottom-up evaluation. Therefore, we only present the top-down approach.

Algorithm Top-Down Retrieval;
Given an object query Q,
retrieve a set of OIDs specified in the query, in which a nested attribute of class C is used.

1. Compute the query signature S_Q for the query Q.
2. For every entry (sig_i, oid_list_i) of the signature file associated with the target class, compare S_Q with sig_i. If they match, then put the oid_list$_i$ into a set S'.
3. For each oid_list$_i$ in S', retrieve the object, which belongs to C, from the list. If its nested attribute satisfies the search condition, put the first OID of oid_list$_i$, which is the OID of a target object, in the returning set S.

Since all the objects along the path are kept track of in the oid_list of the path signature file, there is no traversal along the path for the top-down evaluation. As soon as a matched signature is found, the object which owns the nested attribute may be directly fetched for verification.

4.2 Update

Tree Signature

To update the signature of a modified object, we have to access all attribute objects of the modified object in order to recompute its signature. If the new signature is the same as the old one, the update process stops. Otherwise, it has to be spread to all of its ancestor objects. Therefore, the two important tasks involving the maintenance of signature files are to compute the signature of an object and to spread the update from the modified object to all of its ancestor objects.

Algorithm compute_signature;
Given the OID of an object O,
compute its signature.

1. Use the OID to retrieve O.
2. Generate the signatures for the primitive attributes.
3. Scan all the signature files of complex attributes to find their signatures.
4. Superimpose the signatures of all of O's attributes.
5. Use OID of O to find the location of its signature in the signature file. Then update the signature.

Algorithm spread;
Given the OID of an object O, which has been updated,
update the signatures of O and its ancestor objects.

1. Call compute_signature to update the signature of O.
2. Stop if the old signature and the new signature are the same.
3. Use the old signature of O to retrieve O's parent objects. (This step will use the retrieval algorithm described in the previous section).
4. Recursively apply spread on all parent objects of O.

For this algorithm, if backward references from O to its parent objects are provided, step 3 will be much more efficient.

Path Signature

Similar to the tree signature, we have to update the signatures of the modified object and its ancestor objects to maintain the accuracy of the signatures. The maintenance of signatures is divided into two tasks: to compute the signature of an object and to spread the update of the signature from the modified object to its ancestor objects.

Algorithm compute_signature;
Given the OID of an object O,
compute its signature.

1. Use the OID to retrieve O.
2. If there exists a nested object, O', for the complex attribute on the path, scan the signature file of O' to get the signature of O'.

3. Generate the signatures for the remaining attributes by hashing.

4. Superimpose the signature of O' (if it exists) and the signatures of the other attributes to form the new signature for O.

5. Add O into the oid_list of O' and use it as the oid_list for O's signature entry.

6. Use OID of O to find the location of its signature in the signature file. Then update the signature.

Like the tree signature update, the address of signature may be stored in the object to facilitate direct access of the signature.

The next algorithm is to spread the update of signatures to the classes along the path.

Algorithm spread;
Given a path of classes, $C_1, C_2, ..., C_n$,
update an object O which belongs to class $C_i, 1 \leq i \leq n$.

1. Call compute_signature to update the signature of O.

2. If $i > 1$, scan through the signature file of class C_{i-1} to find a set of parent objects which reference O. It can be done by checking if O is in the oid_list or not.

3. Recursively apply spread on all parent objects of O.

If backward references between objects to their parent objects are provided, the scan of signature file for parent objects may be avoided.

5 COST MODELS

In this section we formulate a cost model to estimate the storage overhead and performance of the tree signature and path signature. The following is the parameters and assumptions used in the cost model.

- C: the number of classes in the database.
- H: the height of the aggregation hierarchy for the database.
- N: the average number of object attributes in a class.
- M: the average number of value attributes in a class.
- Q: the average number of objects in a class.
- S: the size of a signature.
- I: the size of an OID.
- P: the average size of an object.
- A: the average number of parents of an object.
- E: the average size of an entry in the signature files.
- K: the average size of a signature file.
- R_t: the average matching rate of a query signature against a tree signature file.
- R_p: the average matching rate of a query signature against a path signature file.

The unit of size used above is disk page.

5.1 Tree Signature

S is the size of a signature and I is the size of an OID, so $E = S + I$. Thus, the average size of a tree signature file $K = E * Q = (S + I) * Q$.

STORAGE COST: The storage overhead for the database is one signature file for each class in the database. Thus, the storage cost is $K * C$.

RETRIEVAL COST: For a given query, we assume that the length of the path between the target class and the nested attribute is L. Then the number of disk page accesses is $K + R_t * Q * L * P$.

UPDATE COST: For an update operation, the cost is for the retrieval of the object and the update of the signatures of the modified object and its ancestors. The analysis for recomputation of the signature of an object is as follows.

1. To retrieve the object O, P pages of disk access are required.

2. To get the signatures of O's object attributes, $N * K$ disk page accesses are required.

3. To write the signature of O back to its signature file, $K + 1$ pages are required, where K page accesses for finding the location of O's signature and 1 page for writing the updated signature back to the disk.

Therefore, $(N + 1) * K + P + 1$ page accesses are required to compute a signature.

The analysis for spreading the update of a signature to its ancestor objects is as follows. The average number of ancestor objects for an object is $A^{H/2}$. To spread the signature recomputation to a parent object of O, the old signature of O is used as the key to match its potential parents' signatures. The operation is like a retrieval with a direct attribute in the searching condition. The cost of disk accesses is $K + R_t * Q * P$.

Thus, the total disk accesses for update is $A^{H/2} * (K + R_t * Q * P + (N + 1) * K + P + 1)$.

5.2 Path Signature

For a path of classes $C_1, C_2, ..., C_d$, we estimate the cost for storage, retrieval, and update.

STORAGE COST: For each object located in the path, there is a signature. For objects located in C_i where $1 \leq i \leq d$, the size of a signature entry is $S + (d - i + 1) * I$. There are Q objects in a class. Therefore, the storage overhead is $\sum_{i=1}^{d}(S + (d - i + 1) * I) * Q = d * S * Q + Q * I * d * (d + 1)/2$. The average size of a path signature file $K = S * Q + Q * I * (d + 1)/2$.

RETRIEVAL COST: For a given query, the number of disk accesses required is $K + R_p * Q * P$.

UPDATE COST: For an update operation, the cost is for the retrieval of the object and the update of the signatures of the modified object and its ancestor. The analysis for recomputation of the signature of an object is as follows.

1. To retrieve the object O, P pages of disk access are required.

2. To get the signature of its nested attribute, K page accesses are required.

3. To write the signature of O back to the signature file, $K+1$ pages are needed, K page accesses for finding the location of O's signature and 1 page access for writing the updated signature back to the signature file.

Therefore, a total of $2*K+P+1$ pages are required to compute a signature.

The analysis for spreading the update of a signature to the ancestor objects is as follows. The average number of ancestor objects for an object is $A^{d/2}$. To spread the signature recomputation to the parent objects of O, we check if O is in the oid_list of its potential parents, which requires K pages of disk accesses. Thus, the total number of disk accesses for update is $A^{d/2}*(3*K+P+1)$.

6 CONCLUSION

In this paper, we introduce two signature file methods to support the evaluation of queries on nested objects in OODBS. The filtering capability of our signature schemes will reduce the actual secondary storage accesses and thus improve the performance of object retrieval. Comparing to other index schemes, our approaches have a much elegant file structure and much less storage overhead.

The tree signature scheme provides a more general support for queries involving any attributes in the database. To provide equivalent support for arbitrary queries, other index schemes have to create indexes on every nested attributes, which is not feasible due to large storage overhead.

Instead of trying to index all of the attributes in a databases, the path signature scheme pays more attention on attributes in a given path of classes. Every object in the path is indexed by a signature. Since there are less attributes involved, the path signature scheme is more effective on filtering unqualified objects. In the path signature scheme, all nested objects of an object are associated with the path signature. This method reduces object accesses at the expense of some small storage overhead.

The signature schemes not only can support queries with nested attributes, but also can support queries against single or multiple classes in an inheritance hierarchy. Although we do not address the issue of storing a collection of values in an attribute, the signature schemes can easily be extended to create signatures for the collection attributes.

REFERENCES

[1] E. Bertino, "An Indexing technique for object-oriented databases," *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, 1991, 160–170.

[2] E. Bertino & W. Kim, "Indexing techniques for queries on nested objects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, June 1989, 196–214.

[3] O. Deux et al., "The Story of O_2," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, 91–108.

[4] W. Kim, *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA, 1990.

[5] W. Kim, K.-C Kim & A. Dale, "Indexing techniques for object-oriented databases," in *Object-Oriented, Concepts, Databases, and Applications*, W. Kim & F.H. Lochovsky, eds., Addison-Wesley, Reading, MA, 1989, 371–394.

[6] W. Kim et al., "Integrating an object-oriented programming system with a database system," *Proceedings of OOPSLA*, 1988, 142–152.

[7] D. Maier & J. Stein, "Indexing in an object-oriented DBMS," *Proceedings of International Workshop on Object-Oriented Database Systems*, 1986, 171–182.

[8] D. Maier & J. Stein, "Development and implementation of an object-oriented DBMS," in *Readings in Object-Oriented Database Systems*, S. Zdonik & D. Maier, eds., Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990, 167–185.

[9] C.S. Roberts, "Partial-match retrieval via method of superimposed codes," *Proc. IEEE*, Vol. 67, No. 12, Dec. 1979, 1624–1642.