Towards a Realistic Model of Indices in Object Bases*

Boris Novikov
Institute for Mathematics and Mechanics, University of St.-Petersburg
E-mail: boris@orlab.niimm.spb.su

Abstract

A comparison of several indexing techniques for complex object in object-oriented database systems is presented, based on assumption of non-uniform distribution of attribute values. The cost estimations show that conventional ordered indices, especially nested indices with object skeletons, can provide better performance than signature files.

1 Introduction

During the last decade, the object-oriented database management systems become common and proved their advantages and usefulness in several application areas, especially in relatively new for databases. Early research prototypes were followed by several commercial [8, 16, 4], as well as excellent public domain systems [5]. Many commercial relational systems are rapidly evolving to incorporate important features of object-oriented systems, e.g. powerful capabilities of data modelling and support for complex data structures and integrity constrains.

However, while object-oriented systems are extremely efficient in navigational type of data querying, their lack of ability to process efficiently large amounts of data to evaluate complex queries based on attribute values is also widely recognized (see, for example, [12]). More specifically, pure object databases are not well suited to support powerful high-level query languages inherent to relational and post-relational systems. Actually, to introduce any querying facility it is necessary to relax one of the major principles of the object paradigm, namely encapsulation (which is normally done practically in all current systems.

The indexing techniques developed to support queries in relational systems are usually not applicable in the object environment due to

- complex structure of the objects implies that indices should be suitable to support relatively long pointer traversing
- values may be computed with methods, rather than stored as attributes, which significantly increases the frequency and the cost of index updates
- the presence of set-valued attributes makes set comparison operations more frequent.

Several data structures were proposed to support the value-based querying in the object-oriented databases, including join indices [20], access support relations [12], nested indices [2], and signature files [10]. Many of these structures were introduced for completely different environments and their relevance and efficiency for support of object querying is not obvious.

Therefore, it is important to evaluate the behavior of different indexing structures in the object-oriented database systems. An analytical model of this kind was described and used in [11]. However, this model used an incorrect estimation of the expected query selectivity. Estimations in [17] show that, under assumptions of this model, access support relations and nested indices outperform signature files in almost any circumstances.

The estimations of the relative efficiency of different indexing techniques are very important for query optimization: unlike relational systems, the evaluation of the attribute values in the object-oriented database may have significant cost, and therefore should be carefully counted as important for estimation of the query execution plan costs.

However, the same estimations in [17] show that the probability to obtain non-empty result of a query is extremely low, in assumption of uniform distribution of the attribute values (same assumption is used in other papers, including [11]). Obviously, this is not the case in the real-world systems: normally queries return meaningful non-empty results.

The purpose of the on-going research described here is to study the applicability of different indexing techniques proposed for object bases under various assumptions on static and dynamic characteristics of the database. In this paper an attempt to define a more realistic model for queries to the object base with set-valued attributes. To make the things trackable, these estimations are based on a simplified object model.

The cost model of [11, 17] is enhanced to use this distribution instead of uniform, is defined. Finally, different indexing techniques are evaluated using this enhanced model.

The results of both [11] and [17] show that the assumption that attribute values are uniformly distributed is completely unrealistic, because it implies that probability to get non-empty query results is very close to zero.

The rest of the paper is structured as follows. Next section 2 describes the object model and query types, section 3 defines different index structures for complex objects. In section 4 the cost model, retrieval algorithms are presented and discussed. The section 5 contains results of comparison and conclusions.

*This work was partially supported by Russian Foundation for Basic Research under grant 95-01-00636, and UrbanSoft Ltd. under contract 35/95.

Proceedings of the International Workshop on Advances in Databases and Information Systems. Moscow, June 27–30, 1995.

2 The Object and Query Model

In an object-oriented database scheme, real-world entities are represented by only one modelling concept: the object.

Several different object models were proposed during past years. These models highlight various aspects of data modelling and refine the object-oriented paradigm from semantical point of view, resulting in latest standardization activities [18, 6].

However, most of differences between these models are irrelevant to the purposes of this research, which is based on generalized complex object model similar to the models described in [12] and in [20]. Similar models are also used in [14, 9].

Actually all these models completely ignore the behavioral aspect of object-oriented paradigm, and are equally suitable for analysis of systems using the concepts of nested relations [19, 7].

The most interesting aspects of this model concerning indexing to support set operations are:

object identity each object instance has as identity that is invariant throughout its lifetime and is invisible to user.

type constructors, i.e. tuple and set constructors, are used to recursively build complex object types.

object references are stored unidirectional, conforming to almost all published object models.

Complex objects are defined as follows:

1. Integers, strings, BLOBs etc. are considered as *scalar objects.*

2. If $O_1, O_2, \ldots O_n$ are objects and $a_1, a_2, \ldots a_n$ are attribute names, then

$$[a_1 : O_1, a_2 : O_2 \ldots a_n : O_n]$$

is an object called *tuple object*.

3. If $O_1, O_2, \ldots O_n$ are objects, then

$$\{O_1, O_2 \ldots O_n\}$$

is an object that is called *set object*.

It is assumed that any object may be included into several sets, and that sets are represented via object references based on object identity.

In many cases it is necessary to assume that objects in any set are of same *type*. However, we try to avoid this assumption where possible, to allow sets of heterogeneous objects to be represented in this model.

Some of the indexing techniques discussed below can support more specific features of the object-oriented paradigm, e.g. inheritance (class–subclass relationship), which is commonly considered as orthogonal to the complex object structure.

To take into account the inheritance, it is also necessary to define in more details the storage model used for representation of objects (rather than index structures only). There are two major approaches to store objects in the presence of inheritance:

- all attributes of the object may be stored together
- inherited attributes are stored separately within the object representation in the class from which they are inherited (sometimes this representation is called *distributed storage model* [21]).

The storage models described above imply different clustering and have therefore different performance characteristics, as discussed in [9].

As an example, the following database scheme will be considered:

```
disk:[serial-no: string,
     publisher: string,
     location: string
     contents: {items}]
item:[place_where_recorded : string,
     comp: composition,
     played_by: {interpreter}]
composition:[ name: string,
     author: person,
     parts: {part}]
interpreter:[interp: person,
     function: string,
     instrument: string]
person:[name: string,
     . . . ]
```

The queries to this database will normally return locations of disks containing records with desired properties. Nowadays a complete implementation of this system should select required disk in the juke box and play it, using multimedia facilities. However, this aspects of implementation are beyond the scope of this paper.

It is important to note that set comparisons may be necessary for certain queries even if sophisticated algebraic operations are not included in the query language. For example, simple set-inclusion queries are very common in the complex object environment, as described in [11].

In the example database defined above, we may query for disks containing recordings by certain interpreters, i.e. select all disks with records for which the set of interpreters contains the specified set (*superset query*).

A search for disk with only specified interpreters provides an example of *subset query*.

3 Index Structures

In this section several indexing structures that were proposed to support the structure of complex objects are briefly described.

3.1 Signature Indices

The signature indices, sometimes called superimposed coding, were introduced and analyzed in [10] to support indexing for text retrieval. For description of the scheme, it is assumed that objects to be indexed are identified by *sets of index terms* (words in the initial application domain), and queries are represented as sets of terms as well. For each term, a *signature* is constructed as a row of bits (of same length F for all terms). In addition, a number of "1"'s is equal to m (same value for all terms).

To construct term signature, one can use a procedure based on hash functions. e.g. use the values of a series of hash functions as bit position numbers, until specified number m of different bit positions is reached.

To form the signature for a set or a query, all signatures of the terms included into this set (or query) are OR'ed into one bit row of same length F. The use of signature for filtering is based on the fact that

$$S_1 \subseteq S_2 \rightarrow \text{sig}(S_1) \leq \text{sig}(S_2) \quad \text{(bitwise)}.$$

For example, if evaluation of a query Q requires to find all objects that include all terms of query (i.e. $Q \subseteq S$, only objects satisfying $\text{sig}(Q) \leq \text{sig}(S)$ need to be considered).

An important undesirable property of signatures is possibility *false drops*, which may significantly affect the effectiveness of the technique. The probability of false drop is reduced when F increases. Unfortunately, the size of index and time necessary to scan it are also increased in this case.

The signatures of individual objects may be arranged in several ways. In most cases, the best search results may be obtained with bit-sliced multiple block-signature scheme [13]. However, multiple-block superimposed coding performs very poorly on any database updates: actually, it is necessary to re-built the index for each update. Therefore, the deferred update technique may be used to amortize the cost of updates.

Recently the possibility to use signature files as indices for object-oriented databases were studied in [11], based on ordinary and bit-sliced signature files. The index consists of signatures of all sets contained in the objects, accompanied with corresponding OIDs. The performance characteristics for this structure are different from that for text retrieval systems, as shown in [17].

3.2 Nested Indices

The indexing techniques to support search queries on nested complex objects were proposed in [2]. This kind of indices is based on traditional one-dimensional access method, such as B-tree. An index entry consists of a nested attribute value (obtained via certain path), used as a key, and a set of OIDs of all objects that refer to (or contain) this value through specified path.

The performance of this data structure was proposed and studied in [2] for navigational operations in object-oriented databases. In [11], the indices for nested objects are compared with ordinary and bit-sliced signature indices.

The structure of nested indices has known weaknesses, namely, high update cost [21] and lack of support for class hierarchy. Therefore, few enhancements were proposed. In [3, 1] the structure of nested attribute index is augmented with OID index that provides fast access to the list of parents for any object listed in the attribute index.

A further enhancement of the nested index structure is proposed in [15]. This structure, called *object-skeleton* approach, consists of the following components:

- an index on attribute values
- a network of OIDs, representing the complex object structure
- the database containing descriptive information (i.e. object representations with attribute values included).

The advantage of this structure is that navigational network consist of very small records that may be clustered to support fast path traversing, both in complex object structure and class hierarchy, while the attribute values may be clustered on per-object basis. Additional advantage of the object skeletons is that this structure can be used instead of address table, providing a combination of fast physical OID (pointing to skeleton) with flexible storage management for the rest of object data.

3.3 Access Path Relations

The access path relations are described in [12]. This structure is intended to support both navigation and retrieval in object-oriented databases and is defined as follows.

Let $O_0, O_1, O_2, \ldots, O_n$ be the database objects. The expression $O_0.A_1.A_2 \ldots A_n$ is a *path* iff for all $1 < i < n$ one of the following holds:

- Object O_{i-1} is $[\ldots, A_i : O_i, \ldots]$, i.e. the tuple constructor containing attribute A_i.
- Object O_{i-1} is $[\ldots, A_i : \{\ldots\}, \ldots]$, i.e. the tuple constructor containing set-valued attribute A_i.

The latter case is of primary interest for the purposes of this paper.

Given a homogeneous set of paths, an access path relation can be constructed as follows: the tuple of this relation contains attribute values for non-set attributes in the path and object identifiers for set-valued attributes in the path.

Access paths may contain significant amounts of redundant data. As it is shown in [12], access path relations can be decomposed into set of binary and ternary (for set-valued attributes) relations. To reconstruct full paths, relational natural join operation can be applied to decomposed relations.

However, full normalization of access path relation will result in large number of join operations necessary to process it, eliminating any advantage of the structure as access accelerator.

If other (outer, left-outer, or right-outer) natural joins are applied to decomposed relations, the result will contain partial or incomplete paths as well.

The primary purpose of decomposition, is, however, to provide sharing of path segments between different access path relations.

The access relations seem to be very promising not only for navigation operations support, but for set operations as well, because the full power of relational operations can be used for this purpose.

Similar structures were discussed, among others, in the paper on implementation of complex objects [20].

4 The Cost Model

The cost model used here is basically the same as in [17], but with relaxed assumption of uniformity of distribution of the attribute values. Instead, empirical data on value distribution were used for simulation of the indexing behavior. It is assumed that a kind "20-80" rule is applicable in these environment, i.e. the small percentage of the values of the attributes that are most frequent in the database are also frequently used in the queries (user actually knows what is stored in the database).

The typical results of experimental measurements of the value frequencies are shown graphically on the fig. 1.

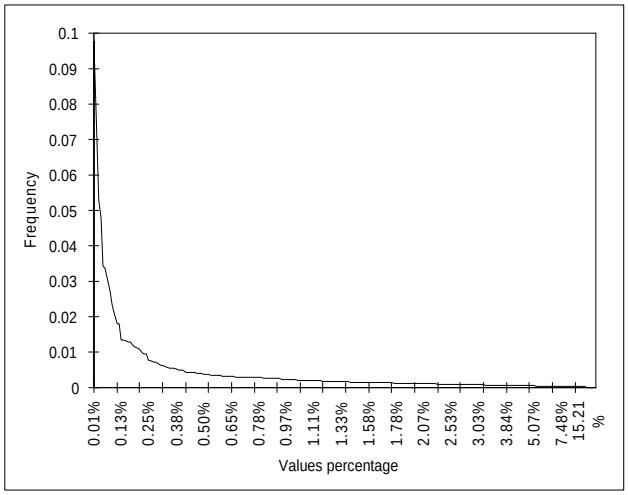

Figure 1: Value frequencies

Another possibility is high correlation of different attribute values included into the same set-valued attribute, i.e. assumption that

normally there is a significant skew of attribute values. However, this possibility is ignored in this research: it is assumed that different values of the members of a set attribute are independent (certainly not always true for example database of this paper).

It is assumed that, for each value $v \in V$ the relative frequency f_v (number of occurrences of v divided by total number of occurrences for all $v \in V$) and selectivity s_v (defined as a percentage of objects that contain this value, through the chain of nesting).

The generalization of the cost model defined in [11] and [12] is used to compare different indexing techniques.

Other parameters of the model are basically same as defined in [17] and are listed in table 1.

Symbol	Definition
N	Total number of objects
P	size of disk page
l_o	size of an OID
V	cardinality of the set domain
O_n	number of OIDs per page
S_{OID}	size of OID file in pages
T	number of page accesses per object

Table 1: Model parameters

To obtain a cost measure, we assume that the query evaluation is performed in the following steps:

1. The initial set of object identifiers (OIDs) $S0_Q$ is calculated using the index. This set contains all OIDs relevant to query Q, and possibly some additional OIDs, due to imprecision of either index or algorithm used to build $S0_Q$.
2. The OIDs of objects not relevant to the query are filtered out of the set $S0$, using either index or direct access to objects.
3. The relevant objects are retrieved and returned to the querying application.

Therefore, the cost of the query may be expressed as follows:

$$total\_cost = index\_cost + filter\_cost + retrieval\_cost.$$

For many cases, some of these costs may be equal to 0. for example, in many cases the only way of filtering is retrieval of objects, and in this case the retrieval cost is already included into cost of filtering. For some algorithms, the set $S0$ may be precise, and, therefore, the cost of filtering will be 0.

However, even in the cases when the index structure provides sufficient information to obtain precise $S0$, it may be reasonable to generate only approximate $S0$ and then use filtering process, because for small sizes of the set filtering may be less expensive than index search, even if access to the stored objects is required.

In this research we are primarily interested in set inclusion queries. The expected number of objects relevant to the query Q may be calculated as

$$E_Q = N * \prod_{v \in Q} s_v$$

and decreases very quickly when the cardinality of the query set q increases.

Since the cost of all access methods in question depends highly on I/O cost, the estimation of cost will be actually estimation of the quantity of disk page accesses.

We assume that direct access to the object via OID requires at most 1 disk access. This is definitely true for physical (location-based) OIDs, and is a reasonable assumption for logical (location-independent) OIDs, if the perfect hashing scheme is used for the object placement. Therfore, the cost of retrieval of objects relevant to query is proportional to the number of the objects, provided that this number is small relative to the total number of the objects in the database. In most cases below, this applies also to the cost of filtering phase, because filtering is performed via the object retrieval.

4.1 Signature Files

The behavior of signature indices created over the object database differs significantly from that for full-text databases due to the differences in quantitative parameters of databases and queries. In the textual databases, the signature for a document is composed from signatures of several terms in the text, while the cardinality of set-valued attributes is relatively small in our study.

The major limitation of the signature indices in the text retrieval applications is high probability of *false drops*, that is, imprecision of indexing. The probability of false drops in object bases is estimated in [17] and are reproduced on the fig. 2.

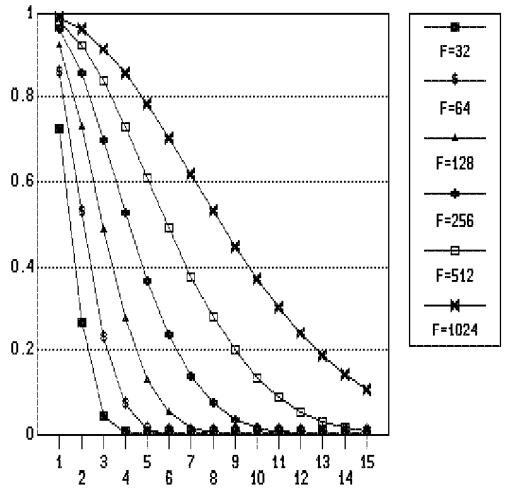

Figure 2: The false drop probabilities

Due to small number of relevant objects, the unconditional probability of false drops is negligible even for short signatures. On the other hand, the cost of index search is relatively high, because, regardless of the query set size, the whole signature index should be scanned to obtain the set $S0$.

This cost may be significantly reduced when the signature file is represented as a set of bit-slices, because only bit-slices corresponding to 1-bits in query signature should be scanned. Further, due to small expected number of relevant objects, it is possible to process only few first bit-slice files to obtain sufficiently small set $S0$ (possibly with large relative amount of false drops to be filtered out).

The total cost of query evaluation can be estimated as

$$C_{slice} * m + C_{obj} = \left\lceil \frac{N}{Pb} \right\rceil * m + TA + TF_m(N - A)$$

where F_m is the false drop probability for m-bit signatures.

The dependency of the total cost from the number of processed bit slices m is shown on fig. 3.

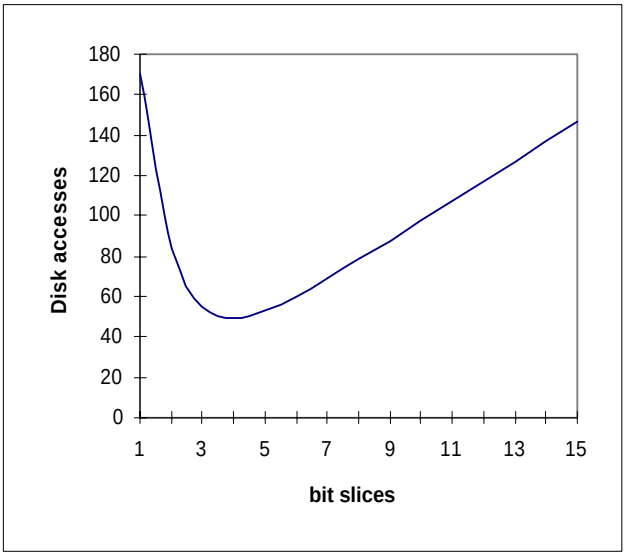

Figure 3: Bit-slice signature costs

The optimal number of bit slices to be processed depends on the database characteristics, but, as shown in [11], normally lies in the range of 2–5 and may be estimated statically provided that database statistics is available.

4.2 Ordered Indices

The essential part of all indexing schemes (except signature files) compared in this study is an ordered index with attribute values as keys. For all structures, this index can provide all data necessary for precise calculation of the set of relevant OIDs, thus eliminating the filtering phase of query evaluation.

However, the cost of index search is approximately proportional to the cardinality of query set and grows linearly when this cardinality increases. At the same time, the cardinality of resulting set decreases exponentially, therefore, it is reasonable to restrict index search to a subset of query set.

The index search may start from the least frequent value from the query set, then the resulting $S0$ can be further restricted using next least frequent value, while the cost of filtering the set $S0$ is greater than the cost of index search for remaining values in the query set. Unfortunately, the cost of obtaining the frequencies for particular values in the query set is comparable with that of index search, therefore, the algorithm described above cannot be practically implemented.

Therefore, we assume that the first and subsequent values are selected randomly from the query set. The expected selectivity can be estimated as

$$\sum_{v \in Q} s_v * p_v$$

and the expected size of the set of OIDs still decreases rapidly when the cardinality of Q grows.

Obviously, the performance of this scheme may be improved, if the frequencies of a small number of most frequent values are stored in the main memory.

4.2.1 Nested Indices

The discussion above directly applies to the structure of nested indices. In the following comparison we assume that the number of attribute values to be searched in the index is determined statically during the query optimization phase based on database statistics.

The cost of ordered (tree) index search can be expressed as $mc\lceil \log_b(N) \rceil$, where b is the average number of index entries per block and c is the number of different object classes in the inheritance hierarchy.

The filtering phase requires costly navigation through complex object hierarchy: each move along the pointer requires a disk access.

4.2.2 Nested Inherited Indices

The structure of nested inherited index differs from (ordinary) nested index only if indexed objects may belong to several different classes in the class hierarchy. In this case, the advantage of the inherited nested index is that the cost of the index search phase is divided by the number of classes in the hierarchy. However, additional cost is necessary to access the secondary (OID) index, instead of searching in several indices in previous case.

The cost of filtering is the same, because there is no difference in the structure.

4.2.3 Object Skeletons

The structure of object skeletons provides fast possibility to navigate through both complex object structure and inheritance hierarchy. Typically, all data describing complex object will fit into single page. Therefore, the cost of filtering phase is less than that for both types of nested indices. Therefore, the estimations for the search cost are $m\lceil \log_b(N) \rceil$, and $2S$ disk accesses for filtering phase, because one access is necessary to obtain the object skeleton and another to check the attribute value.

4.2.4 Access Path Relations

The structure of access paths belongs to ”higher level” than other considered in this study, and performance may depend significantly on actual implementation.

It is assumed that access path relation is represented as a pair of ordered indices (with first and last attributes of the relation a s the keys. Therefore, any search for particular attribute value requires logarithmic number of disk accesses (depending on the size of access relation).

The major advantage of the access path relations is that filtering can be performed using the relations without retrieving the complex objects in the database.

5 Comparison and Conclusions

To obtain the empirical data on the value distribution, a medium-size bibliographic database was used. The abstracts of papers stored in this database were considered as set-valued attributes (i.e. abstracts were treated as sets of words. The obtained frequencies are used as probabilities in a simple analytical model similar to described in [17], as well as for simulation experiment with randomly generated queries (taking into account the same frequencies).

The results of total cost estimation for different indexing techniques are represented on fig. 4.

The results of both modelling and simulation clearly show that in most cases in the realistic parameter ranges the conventional indexing techniques based on inverted lists (nested indices, access path relations and similar) provide very good scalability and usually slightly outperform signature files of different type, with few exceptions. The advanced variants of nested indices clearly outperform initial structure.

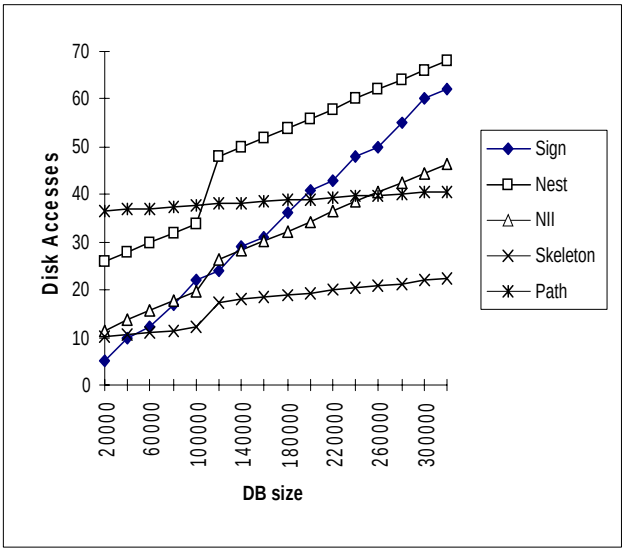

Figure 4: Query costs

The model presented in this paper is built in assumption of "cold" query execution, that is, it is assumed that all data necessary for query processing are read from disk. With rapid growth of memory sizes this assumption also becomes unrealistic and may result in significant performance implications. Even if assumption that the whole database can fit into main memory is still unrealistic, some important parts of database, for example, object skeletons or signature lists are usually small enough to be kept in memory.

Another issue missed here is update costs, which are especially important for creation of indices on method values.

References

[1] E. Bertino. An Indexing Technique for Object-Oriented Databases. In *Proc. 7th Intnl.Conf. on Data Engineering. Kobe, Japan.*, pages 160–170, 1991.

[2] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Data Eng.*, 2(1):196–214, 1989.

[3] E. Bertino and L. Martino. Object-oriented database management systems: Concepts and issues. *IEEE Computer*, 24(4):33–47, April 1991.

[4] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone Object Database Management System. *Communications of the ACM*, 34(10):64–77, 1991.

[5] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. 12 conf. VLDB*, pages 91–100, 1986.

[6] R. G. G. Cattell. *The Object Database Standard OMDG-93*. Morgan-Kaufmann, 1993.

[7] A. Deshpande and D. Gucht. A storage structure for nested relational databases. In *Nested relations and complex objects in databases*, volume 361 of *Lect. Notes in Comp. Sci.*, pages 69–83, 1989.

[8] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–49, 1991.

[9] H. Dombrowska, I. Kaprizkina, and B. Novikov. Representation of the SYNTHESIS data structures in the object store. In *Proc. of the workshop on advances in databases and information systems - ADBIS'93*, pages 60–68, Moscow, May 22–24 1993.

[10] C. Faloutsos and S. Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. on Database Systems*, 4(2):267–288, 1984.

[11] Yoshiharu Ishikawa, Hiroyuki Kitagawa, and Nobuo Ohbo. Evaluation of signature files as set access facilities in OODBs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, volume 22:2 of *SIGMOD Rec.*, pages 247–256, Washington, DC, May 26–28 1993.

[12] Alfons Kemper and Guido Moekotte. Access support relations: an indexing method for object bases. *Inf. Syst. (Oxford)*, 17(2):117–145, 1992.

[13] A. Kent, R. Sacks-Davies, and K. Ramamohanarao. A superimposed coding scheme based on multiple block descriptor files for indexing very large databases. In *Proc. 14 conf. VLDB*, pages 351–359, 1988.

[14] S. Khoshafian, M. Franklin, and M.J. Carey. Storage management for persistent complex objects. *Inf. Syst.*, 15(3):303–320, 1990.

[15] Kien A. Hua and Chimnoy Tripathy. *Object Skeletons:* An Efficient Navigation Structure for Object-Oriented Database Systems. In *Proc. 10th Intnl.Conf. on Data Engineering. Huston, TX.*, pages 508–517, 1994.

[16] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[17] B. A. Novikov. Indices for set-theoretical operations in object bases. In *Proc. of the Intnl. Workshop on Advances in Databases and Information Systems - ADBIS'94*, pages 208–216, Moscow, May 23–26 1994.

[18] OMG and Xopen. *The Common Object Request Broker: Architecture and Specification*. Object Management Group and X/Open, Framingham, MA and Reading Berkshire, UK, 1992.

[19] M. Scholl, S. Abiteboul, F. Banchilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A database machine based on nested relations. In *Nested relations and complex objects in databases*, volume 361 of *Lect. Notes in Comp. Sci.*, pages 27–49, 1989.

[20] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation technique of complex objects. In *Proc. 12 conf. VLDB*, pages 101–110, 1986.

[21] Y.S. Rahim Yaseen and H. Lam. An Extensible Kernel Object Management System. In *Proc. of OOPSLA'91.*, pages 247–263, 1991.