Efficient Use of Signatures in Object-Oriented Database Systems

Kjetil Nørvåg

Department of Computer and Information Science
Norwegian University of Science and Technology
7491 Trondheim, Norway
email: noervaag@idi.ntnu.no

Abstract. Signatures are bit strings, generated by applying some hash function on some or all of the attributes of an object. The signatures of the objects can be stored separately from the objects themselves, and can later be used to filter out candidate objects during a perfect match query. In an object-oriented database system (OODB) using logical object identifiers (OIDs), an object identifier index (OIDX) is needed to map from logical OID to the physical location of the object. In this paper we show how signatures can be stored in the OIDX, and used to reduce the average object access cost in a system. We also extend this approach to transaction time temporal OODBs (TOODB), where this approach is even more beneficial, because maintaining signatures comes virtually for free. We develop a cost model that we use to analyze the performance of the proposed approaches, and this analysis shows that substantial gain can be achieved.

1 Introduction

A signature is a bit string, which is generated by applying some hash function on some or all of the attributes of an object.[1] When searching for objects that match a particular value, it is possible to decide from the signature of an object whether the object is a possible match. The size of the signatures is generally much smaller than the size of the objects themselves, and they are normally stored separately from the objects themselves, in signature files. By first checking the signatures when doing a perfect match query, the number of objects to actually be retrieved can be reduced.

Signature files have previously been shown to be an alternative to indexing, especially in the context of text retrieval [1, 6]. Signature files can also be used in general query processing, although this is still an immature research area.

The main drawback of signature files, is that signature file maintenance can be relatively costly. If one of the attributes contributing to the signature in an object is modified, the signature file has to be updated as well. To be beneficial, a high read to write ratio is necessary. In addition, high selectivity is needed at query time to make it beneficial to read the signature file in addition to the objects themselves.

[1] Note that *signatures* are also often used in other contexts, e.g., function signatures and implementation signatures.

In this paper, we first show how signatures can be incorporated into traditional object-oriented databases (OODB). Second, we show how they can be used in *transaction time temporal OODBs*, with only marginal maintenance cost.

Every object in an OODB is uniquely identified by an object identifier (OID). To do the mapping from logical OID to physical location, an OID index (OIDX), often a B-tree variant, is used.[2] The entries in the OIDX, which we call *object descriptors* (OD), contains the physical address of the object. Because of the way OIDs are generated, OIDX accesses often have low locality, i.e., often only one OD in a particular OIDX leaf node is accessed at a time. This means that OIDX lookups can be costly, but they have to be done every time an object is to be accessed (as will be explained later, the lookup cost can be reduced by employing OD caching). OIDX updates are only needed when object are created, moved, or deleted. It is not necessary when objects are updated, because updates to object are done in-place, so that the mapping information in the OIDX is still valid after an object have been updated.

Our approach to reduce the average access cost in the system, is to include the signature of an object *in the OIDX itself*. This means that the OD now also includes the signature, in addition to the mapping information. When we later do a value based perfect match search on a set, we can in many cases avoid retrieving the objects themselves, checking the signature in the OD is enough to exclude an object during the search. The OD will have to be retrieved anyway, because it is needed to find the physical location of the object, so there is no additional cost to retrieve the OD, compared to not using signatures. Storing the signature in the OIDX increases the size of the OD, and the size of the OIDX, and makes an OIDX update necessary every time an object is updated, but as we will show later in this paper, in spite of this extra cost, it will in most cases be beneficial.

A context where storing signatures in the OIDX is even more interesting, is transaction time temporal OODBs (TOODB). In a TOODB, object updates do not make previous versions unaccessible. On the contrary, previous versions of objects can still be accessed and queried. A system maintained timestamp is associated with every object version. This timestamp is the commit time of the transaction that created this version of the object. In a non-temporal OODB, the OIDX update would not be necessary if we did not want to maintain signatures. In a TOODB, on the other hand, *the OIDX must be updated every time an object is updated*, because we add a new version, and the timestamp and the physical address of the new version need to be inserted into the index. As a result, introducing signatures only marginally increases the OIDX update costs. Because of the low locality on updates, disk seek time dominates, and the increased size of the ODs is of less importance. *With this approach, we can maintain signatures at a very low cost, and by using signatures, one of the potential bottlenecks in an TOODB, the frequent and costly OIDX updates, can be turned into an advantage!*

The organization of the rest of the paper is as follows. In Sect. 2 we give an overview of related work. In Sect. 3 we give a brief introduction to signatures. In Sect. 4 we describe indexing and object management in a TOODB. In Sect. 5 we describe how

[2] Some OODBs avoid the OIDX by using physical OIDs. In that case, the OID gives the physical disk page directly. While this potentially gives a higher performance, it is very inflexible, and makes tasks as reclustering and schema management more difficult.

signatures can be integrated into OID indexing. In Sect. 6 we develop a cost model, which we use in Sect. 7 to study the performance when using signatures stored in the OIDX, with different parameters and access patterns. Finally, in Sect. 8, we conclude the paper and outline issues for further research.

2 Related Work

Several studies have been done in using signatures as a text access methods, e.g. [1, 6]. Less has been done in using signatures in ordinary query processing, but studies have been done on using signatures in queries on set-valued objects [7].

We do not know of any OODB where signatures have been integrated, but we plan to integrate the approaches described in this paper in the Vagabond parallel TOODB [9].

3 Signatures

In this section we describe signatures, how they can be used to improve query performance, how they are generated, and signature storage alternatives.

A signature is generated by applying some hash function on the object, or some of the attributes of the object. By applying this hash function, we get a signature of F bits, with m bits set to 1. If we denote the attributes of an object i as $A_1, A_2, \ldots, A_n$, the signature of the object is $s_i = S_h(A_j, \ldots A_k)$, where S_h is a hash value generating function, and $A_j, \ldots A_k$ are some or all of the attributes of the object (not necessarily including all of $A_j, \ldots A_k$). The size of the signature is usually much smaller than the object itself.

3.1 Using Signatures

A typical example of the use of signatures, is a query to find all objects in a set where the attributes match a certain number of values, $A_j = v_j, \ldots, A_k = v_k$. This can be done by calculating the query signature s_q of the query: $s_q = S_h(A_j = v_j, \ldots, A_k = v_k)$ The query signature s_q is then compared to all the signatures s_i in the signature file to find possible matching objects. A possible matching object, a *drop*, is an object that satisfies the condition that all bit positions set to 1 in the query signature, also are set to 1 in the object's signature. The drops forms a set of candidate objects. An object can have a matching signature even if it does not match the values searched for, so all candidate objects have to be retrieved and matched against the value set that is searched for. The candidate objects that do not match are called *false drops*.

3.2 Signature Generation

The methods used for generating the signature depend on the intended use of the signature. We will now discuss some relevant methods.

Whole Object Signature. In this case, we generate a hash value from the whole object. This value can later be used in a perfect match search that includes all attributes of the object.

One/Multi Attribute Signatures. The first method, *whole object signature*, is only useful for a limited set of queries. A more useful method is to create the hash value of only one attribute. This can be used for perfect match search on that specific attribute. Often, a search is on perfect match of a subset of the attributes. If such searches are expected to be frequent, it is possible to generate the signature from these attributes, again just looking at the subset of attributes as a sequence of bits. This method can be used as a filtering technique in more complex queries, where the results from this filtering can be applied to the rest of the query predicate.

Superimposed Coding Methods. The previous methods are not very flexible, they can only be used for queries on the set of attributes used to generate the signature. To be able to support several query types, that do perfect match on different sets of attributes, a technique called *superimposed coding* can be used. In this case, a separate attribute signature for each attribute is created. The object signature is created by performing a bitwise OR on each attribute signature, for an object with 3 attributes the signature is $s_i = S_h(A_0)$ OR $S_h(A_1)$ OR $S_h(A_2)$. This results in a signature that can be very flexible in its use, and support several types of queries, with different attributes.

Superimposed coding is also used for fast text access, one of the most common applications of signatures. In this case, the signature is used to avoid full text scanning of each document, for example in a search for certain words occurring in a particular document. There can be more than one signature for each document. The document is first divided into logical blocks, which are pieces of text that contain a constant number of distinct words. A word signature is created for each word in the block, and the block signature is created by OR-ing the word signatures.

3.3 Signature Storage

Traditionally, the signatures have been stored in one or more separate files, outside the indexes and objects themselves. The files contains s_i for all objects i in the relevant set. The sizes of these files are in general much smaller than the size of the relation/set of objects that the signatures are generated from, and a scan of the signature files is much less costly than a scan of the whole relation/set of objects. Two well-know storage structures for signatures are *Sequential Signature Files* (SSF) and *Bit-Sliced Signature Files* (BSSF). In the simplest signature file organization, SSF, the signatures are stored sequentially in a file. A separate *pointer file* is used to provide the mapping between signatures and objects. In an OODB, this file will typically be a file with OIDs, one for each signature. During each search for perfect match, the whole signature file has to be read. Updates can be done by updating only one entry in the file.

With BSSF, each bit of the signature is stored in a separate file. With a signature size F, the signatures are distributed over F files, instead of one file as in the SSF approach. This is especially useful if we have large signatures. In this case, we only have to search the files corresponding to the bit fields where the query signature has a "1". This can reduce the search time considerably. However, each update implies updating up to F files, which is very expensive. So, even if retrieval cost has been shown to be much smaller for BSSF, the update cost is much higher, 100-1000 times higher is not

uncommon [7]. Thus, BSSF based approaches are most appropriate for relatively static data.

4 Object and Index Management in TOODB

We start with a description of how OID indexing and version management can be done in a TOODB. This brief outline is not based on any existing system, but the design is close enough to make it possible to integrate into current OODBs if desired.

4.1 Temporal OID Indexing

In a traditional OODB, the OIDX is usually realized as a hash file or a B^+-tree, with ODs as entries, and using the OID as the key. In a TOODB, we have more than one version of some of the objects, and we need to be able to access current as well as old versions efficiently. Our approach to indexing is to have *one* index structure, containing all ODs, current as well as previous versions.

In this paper, we assume one OD for each object version, stored in a B^+-tree. We include the commit time *TIME* in the OD, and use the concatenation of OID and time, $OID||TIME$, as the index key. By doing this, ODs for a particular OID will be clustered together in the leaf nodes, sorted on commit time. As a result, search for the current version of a particular OID as well as retrieval of a particular time interval for an OID can be done efficiently. When a new object is *created*, i.e., a new OID allocated, its OD is appended to the index tree as is done in the case of the Monotonic B^+-tree [5]. This operation is very efficient. However, when an object is *updated*, the OD for the new version *has to be inserted into the tree.*

While several efficient multiversion access methods exist, e.g., TSB-tree [8], they are not suitable for our purpose, because they provide more flexibility than needed, e.g., combined key range and time range search, at an increased cost. We will never have search for a (consecutive) range of OIDs, OID search will always be for *perfect match*, and most of them are assumed to be to the current version. It should be noted that our OIDX is inefficient for many typical temporal queries. As a result, additional secondary indexes can be needed, of which TSB-tree is a good candidate. However, *the OIDX is still needed*, to support navigational queries, one of the main features of OODBs compared to relational database systems.

4.2 Temporal Object Management

In a TOODB, it is usually assumed that most accesses will be to the current versions of the objects in the database. To keep these accesses as efficient as possible, and to benefit from object clustering, the database is partitioned. Current objects reside in one partition, and the previous versions in the other partition, in the *historical database*. When an object is updated in a TOODB, the previous version is first moved to the historical database, before the new version is stored in-place in the current database.

We assume that clustering is not maintained for historical data, so that all objects going historical, i.e., being moved because they are replaced by a new version, can be

written sequentially, something which reduces update costs considerably. The OIDX is updated *every time an object is updated*.

Not all the data in a TOODB is temporal, for some of the objects, we are only interested in the current version. To improve efficiency, the system can be made aware of this. In this way, some of the data can be defined as non-temporal. Old versions of these are not kept, and objects can be updated in-place as in a traditional OODB, and the costly OIDX update is not needed when the object is modified. This is an important point: using an OODB which efficiently supports temporal data management, should not reduce the performance of applications that do not utilize these features.

5 Storing Signatures in the OIDX

The signature can be stored together with the mapping information (and timestamp in the case of TOODBs) in the OD in the OIDX. Perfect match queries can use the signatures to reduce the number of objects that have to be retrieved, only the candidate objects, with matching signatures, need to be retrieved.

Optimal signature size is very dependent of data and query types. In some cases, we can manage with a very small signature, in other cases, for example in the case of text documents, we want a much larger signature size. It is therefore desirable to be able to use different signature sizes for different kind of data, and as a result, we should provide different signature sizes.

The maintenance of object signatures implies computational overhead, and is not always required or desired. Whether to maintain signatures or not, can for example be decided on a per class basis.

6 Analytical Model

Due to space constraints, we can only present a brief overview of the cost model used. For a more detailed description, we refer to [10]. The purpose of this paper is to show the benefits of using signatures in the OIDX, so we will restrict this analysis to attribute matches, using the superimposed coding technique.

Our cost model focus on disk access costs, as this is the most significant cost factor. In our disk model, we distinguish between random and sequential accesses. With random access, the time to read or write a page is denoted T_P, with sequential access, the time to read or write a page is T_S. All our calculations are based on a page size of 8 KB.

The system modeled in this paper, is a page server OODB, with temporal extensions as described in the previous sections. To reduce disk I/O, the most recently used index and object pages are kept in a *page buffer* of size M_{pbuf}. OIDX pages will in general have low locality, and to increase the probability of finding a certain OD needed for a mapping from OID to physical address, the most recently used ODs are kept in a separate OD cache of size M_{ocache}, containing N_{ocache} ODs. The OODB has a total of M bytes available for buffering. Thus, when we talk about the memory size M, we only consider the part of main memory used for buffering, not main memory used for the

objects, the program itself, other programs, the operating system, etc. The main memory size M is the sum of the size of the page buffer and the OD cache, $M = M_{\text{pbuf}} + M_{\text{ocache}}$.

With increasing amounts of main memory available, buffer characteristics are very important. To reflect this, our cost model includes buffer performance as well, in order to calculate the hit rates of the OD cache and the page buffer. Our buffer model is an extension of the Bhide, Dan and Dias LRU buffer model [2]. An important feature of the BDD model, which makes it more powerful than some other models, is that it can be used with *non-uniform access distributions*. The derivation of the BDD model in [2] also includes an equation to calculate the number N_d of distinct objects out of a total of N access objects, given a particular access distribution. We denote this equation $N_{distinct}(N_d, N)$. The buffer hit probability of an object page is denoted P_{buf_opage}. Note that even if index and object pages share the same page buffer, the buffer hit probability is different for index and object pages. We do not consider the cost of log operations, because the logging is done to separate disks, and the cost is independent of the other costs.

To analyze the use of signatures in the OIDX, we need a cost model that includes:

1. OIDX update and lookup costs.
2. Object storage and retrieval costs.

The OIDX lookup and update costs can be calculated with our previously published cost model [10]. The only modification done to this cost model is that signatures are stored in the object descriptors (ODs). As a consequence, the OD size varies with different signature sizes. In practice, a signature in an OD will be stored as a number of bytes, and for this reason, we only consider signature sizes that are multiples of 8 bits.

The average OIDX lookup cost, i.e., the average time to retrieve the OD of an object, is denoted $T_{\text{oidx\_lookup}}$, and the average time to do an update is $T_{\text{oidx\_update}}$. Not all objects are temporal, and in our model, we denote the fraction of the operations done on temporal objects as P_{temporal}. For the non-temporal objects, if signatures are not maintained, the OIDX is only updated when the objects are created.

6.1 Object Storage and Retrieval Cost Model

One or more objects are stored on each disk page. To reduce the object retrieval cost, objects are often placed on disk pages in a way that makes it likely that more than one of the objects on a page that is read, will be needed in the near future. This is called clustering. In our model, we define the clustering factor C as the fraction of an object page that is relevant, i.e., if there are N_{o_page} objects on each page, and n of them will be used, $C = \frac{n}{N_{o\_page}}$. If $N_{o\_page} < 1.0$, i.e., the average object size is larger than one disk page, we define $C = 1.0$.

Read Objects. We model the database read accesses as 1) *ordinary object accesses*, assumed to benefit from the database clustering, and 2) *perfect match queries*, which can benefit from signatures. We assume the perfect match queries to be a fraction P_{qm} of the read accesses, and that P_A is the fraction of queried objects that are actual drops. Assuming a clustering factor of C, the average object retrieval cost, excluding OIDX

lookup, is $T'_{readobj} = \frac{1}{CN_{o\_page}} T_{readpage}$, where the average cost of reading one page from the database, is $T_{readpage} = (1 - P_{buf\_opage})T_P$. When reading object pages during signature based queries, we must assume we can not benefit from clustering, because we retrieve only a very small amount of the total number of objects. In that case, one page must be read for every object that is retrieved, $T''_{readobj} = T_{readpage}$. The average object retrieval cost, employing signatures, is:

$$T_{readobj} = T_{\text{oidx\_lookup}} + (1 - P_{qm})T'_{readobj} + P_{qm}(P_A T''_{readobj} + (1 - P_A)F_d T''_{readobj})$$

which means that of the P_{qm} that are queries for perfect match, we only need to read the object page in the case of actual or false drops. The false drop probability when a signature with F bits is generated from D attributes is denoted $F_d = (\frac{1}{2})^m$, where $m = \frac{F \ln 2}{D}$.

Update Objects. Updating can be done in-place, with write-ahead logging (but note that in the case of temporal objects, these are moved to the historical partition before the new current version is written). In that case, a transaction can commit after its log records have been written to disk. Modified pages are not written back immediately, this is done lazily in the background as a part of the buffer replacement and checkpointing. Thus, a page may be modified several times before it is written back.

Update costs will be dependent of the checkpoint interval. The checkpoint interval is defined to be the number of objects that can be written between two checkpoints. The number of written objects, N_{CP}, includes created as well as updated objects. $N_{CR} = P_{new}N_{CP}$ of the written objects are creations of new objects, and $(N_{CP} - N_{CR})$ of the written objects are updates of existing objects.

The number of distinct updated objects during one checkpoint period is $N_{DU} = N_{distinct}(N_{CP} - N_{CR}, N_{obj})$. The average number of times each object is updated is $N_U = \frac{N_{CP} - N_{CR}}{N_{DU}}$. During one checkpoint interval, the number of pages in the current partition of the database that is affected is $N_P = \frac{N_{DU}}{N_{o\_page}C}$ This means that during one checkpoint interval, new versions must be inserted into N_P pages. CN_{o_page} objects on each page have been updated, and each of them have been updated an average of N_U times. For each of these pages, we need to write $P_{\text{temporal}}N_U CN_{o\_page}$ objects to the historical partition (this includes objects from the page and objects that were not installed into the page before they went historical), install the new current version to the page, and write it back. This will be done in batch, to reduce disk arm movement, and benefits from sequential writing of the historical objects. For each of the object updates, the OIDX must be updated as well. In the case of a non-temporal OODB, we do not need to write previous versions to the historical partition, and the OIDX needs only to be updated if signatures are to be maintained.

When new objects are created, an index update is needed. When creating a new object, a new page will, on average, be allocated for every N_{o_page} object creation. When a new page is allocated, installation read is not needed. The average object update cost, excluding OIDX update cost:

$$T_{write\_obj} = T_{\text{oidx\_update}} + \frac{N_P T_S(P_{\text{temporal}}N_U CN_{o\_page}) + N_P T_P + \frac{N_{CR}}{N_{o\_page}} T_P}{N_{CP}}$$

Note that objects larger than one disk page will usually be partitioned, and each object is indexed by a separate large object index tree. This has the advantage that when a new version is created, only the modified parts need to be written back. An example of how this can be done is the EXODUS large objects [3].

7 Performance

We have now derived the cost functions necessary to calculate the average object storage and retrieval costs, with different system parameters and access patterns, and with and without the use of signatures. We will in this section study how different values of these parameters affect the access costs. Optimal parameter values are dependent of the mix of updates and lookups, and they should be studied together. If we denote the probability that an operation is a write, as P_{write}, the average access cost is the weighted cost of average object read and write operations:

$$T_{\text{access}} = (1 - P_{\text{write}})T_{\text{lookup}} + P_{\text{write}}T_{\text{update}}$$

Our goal in this study is to minimize T_{access}. We measure the gain from the optimization as: Gain $= 100\left(\frac{(T^{nonopt}_{\text{access}} - T^{opt}_{\text{access}})}{T^{opt}_{\text{access}}}\right)$ where $T^{nonopt}_{\text{access}}$ is the cost if not using signatures, and T^{opt}_{access} is the cost using signatures.

Access Model The access pattern affects storage and retrieval costs directly and indirectly, through the buffer hit probabilities. The access pattern is one of the parameters in our model, and is modeled through the independent reference model. Accesses to objects in the database system are assumed to be random, but skewed (some objects are more often accessed than others), and the objects in the database can be logically partitioned into a number of partitions, where the size and access probability of each partition can be different. With β_i denoting the relative size of a partition, and α_i denoting the percentage of accesses going to that partition, we can summarize the four access patterns used in this paper:

Set	β_0	β_1	β_2	α_0	α_1	α_2
3P1	0.01	0.19	0.80	0.64	0.16	0.20
3P2	0.001	0.049	0.95	0.80	0.19	0.01
2P8020	0.20	0.80	-	0.80	0.20	-
2P9505	0.05	0.95	-	0.95	0.05	-

In the first partitioning set, we have three partitions. It is an extensions of the 80/20 model, but with the 20% hot spot partition further divided into a 1% hot spot area and a 19% less hot area. The second partitioning set, 3P2 resembles the access pattern close to what we expect it to be in future TOODBs. The two other sets in this analysis have each two partitions, with hot spot areas of 5% and 20%.

Parameter	Value	Parameter	Value	Parameter	Value
M	100 MB	S_{obj}	128	C	0.3
M_{ocache}	0.2 M	N_{objver}	100 mill.	D	1
N_{CP}	$0.8N_{\text{ocache}}$	S_{od}	$32 + \lceil \frac{F}{8} \rceil$	P_{new}	0.2
P_A	0.001	P_{write}	0.2	P_{qm}	0.4
P_{temporal}	0.8				

Table 1. Default parameters.

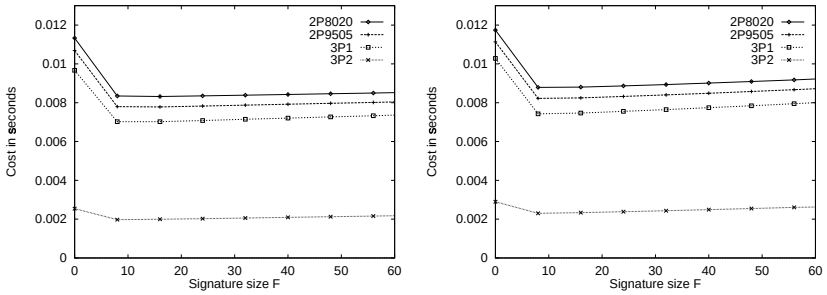

Fig. 1. Cost versus signature size for different access patterns. Non-temporal OODB to the left, temporal OODB to the right.

Parameters We consider a database in a stable condition, with a total of N_{objver} objects versions (and hence, N_{objver} ODs in the OIDX). Note that with the OIDX described in Section 4.1, OIDX performance is not dependent of the number of existing versions of an object, only the total number of versions in the database.

Unless otherwise noted, results and numbers in the next sections are based on calculations using default parameters, as summarized in Table 1, and access pattern according to partitioning set 3P1.

With the default parameters, the studied database has a size of 13 GB. The OIDX has a size of 3 GB in the case of a non-temporal OODB, and 5 GB in the case of a temporal OODB (not counting the extra storage needed to store the signatures). Note that the OIDX size is smaller in a non-temporal OODB, because in a non-temporal OODB, we do not have to store timestamps, and we have no inserts into the index tree, only append-only. In that case, we can get a very good space utilization [4]. When we have inserts into the OIDX, as in the case of a temporal OODB, we get a space utilization in the OIDX that is less than 0.67.

7.1 Optimal Signature Size

Choosing the signature size is a tradeoff. A larger signature can reduce the read costs, but will also increase the OIDX size and OIDX access costs. Figure 1 illustrates this for different access patterns. In this case, a value of $F = 8$ seems to be optimal. This

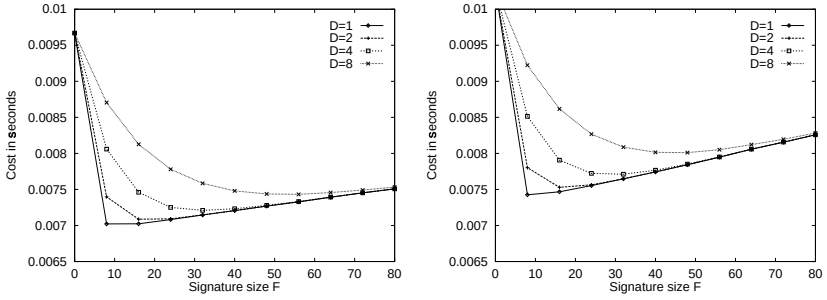

Fig. 2. Cost versus signature size for different values of D. Non-temporal OODB to the left, temporal OODB to the right.

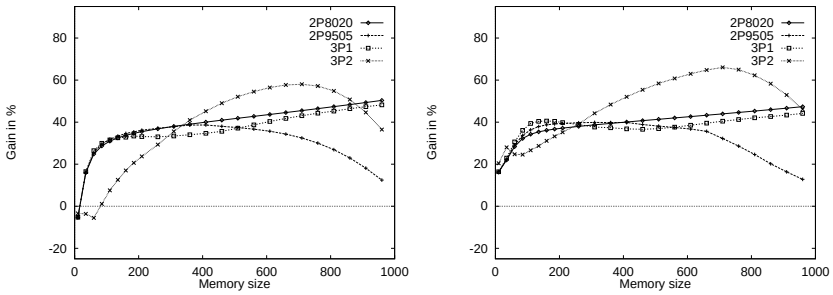

Fig. 3. Gain from using signatures versus memory size, for different access patterns. Non-temporal OODB to the left, temporal OODB to the right.

is quite small, and gives a higher false drop probability than accepted in text retrieval applications. The reason why such a small signature is optimal in our context, is that the size of objects is small enough to make object retrieval and match less costly than a document (large object) retrieval and subsequent search for matching word(s), as is the case in text retrieval applications.

The signature size is dependent of D, the number of attributes contributing to the signature. This is illustrated in Fig. 2. With an increasing value of D, the optimal signature size increases as well. In our context, a value of D larger than one, means that more than one attribute contributes to the signature, so that queries on more than one attribute can be performed later.

In the rest of this study, we will use F=8 when using the default parameters, and use F=16, 32 and 48 for D=2, 4, and 8, respectively.

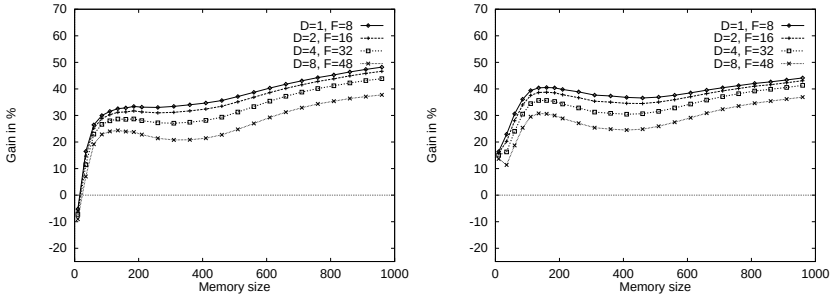

Fig. 4. Gain from using signatures, versus memory size, for different values of D. Non-temporal OODB to the left, temporal OODB to the right.

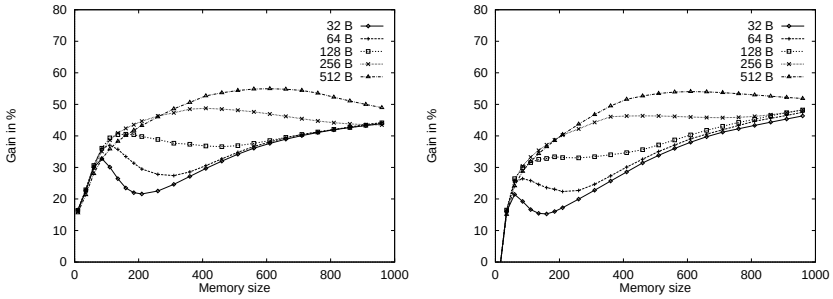

Fig. 5. Gain from using signatures, versus memory size, for different average object sizes. Non-temporal OODB to the left, temporal OODB to the right.

7.2 Gain From Using Signatures

Figure 3 shows the gain from using signatures, with different access patterns. Using signatures is beneficial for all access patterns, except when only a very limited amount of memory is available.

Figure 4 shows the gain from using signatures, for different values of D. The gain decreases with increasing value of D.

7.3 The Effect of the Average Object Size

We have chosen 128 as the default average object size. It might be objected that this value is too large, but Fig. 5 shows that even with smaller object sizes, using signatures will be beneficial. The figure also shows how the gain increases with increasing object size.

7.4 The Effect of P_A and P_{qm}

The value of P_{qm} is the fraction of the read queries that can benefit from signatures. Figure 6 illustrates the gain with different values of P_{qm}. As can be expected, a small

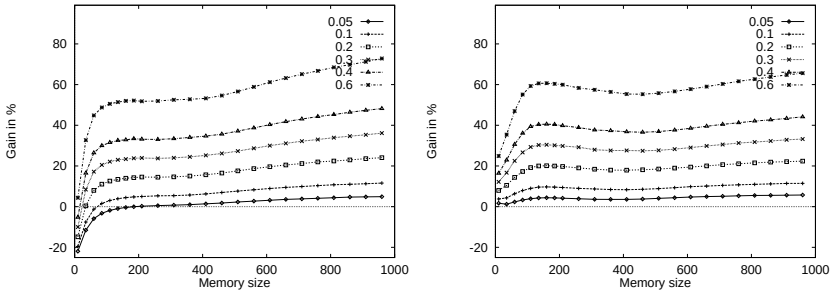

Fig. 6. Gain from using signatures, versus memory size, for different values of P_{qm}. Non-temporal OODB to the left, temporal OODB to the right.

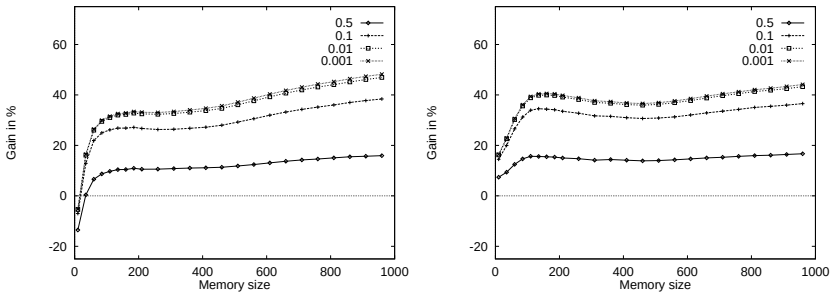

Fig. 7. Gain from using signatures, versus memory size, for different values of P_A. Non-temporal OODB to the left, temporal OODB to the right.

value of P_{qm} results in negative gain in the case of non-temporal OODBs, i.e., in this case, storing and maintaining signatures in the OIDX reduces the average performance.

The value of P_A is the fraction of queries objects that are actual drops, the selectivity of the query. Only if the value of P_A is sufficiently low, will we be able to benefit from using signatures. Figure 7 shows that signatures will be beneficial even with a relatively large value of P_A.

8 Conclusions

We have in this paper described how signatures can be stored in the OIDX. As the OD is accessed on every object access in any case, there is no extra signature retrieval cost. In a traditional, non-versioned OODBs, maintaining signatures means that the OIDX needs to be updated every time an object is updated, but as the analysis shows, it will in most cases pay back, as less objects need to be retrieved.

Storing signatures in the OIDX is even more attractive for TOODBs. In TOODBs, the OIDX will have to be updated on every object update anyway, so in that case, the extra cost associated with signature maintenance is very low.

As showed in the analysis, substantial gain can be achieved by storing the signature in the OIDX. We have done the analysis with different system parameters, access patterns, and query patterns, and in most cases, storing the object signatures in the OIDX is beneficial. The typical gain is from 20 to 40%. Interesting to note is that the optimal signature size can be quite small.

The example analyzed in this paper is quite simple. A query for perfect match has a low complexity, and there is only limited room for improvement. The real benefit is available in queries where the signatures can be used to reduce the amount of data to be processed at subsequent stages of the query, resulting in larger amounts of data that can be processed in main memory. This can speed up query processing several orders of magnitude.

Another interesting topic for further research is using signatures in the context of bitemporal TOODBs.

Acknowledgments

The author would like to thank Olav Sandstå for proofreading, and Kjell Bratbergsengen, who first introduced him to signatures.

References

1. E. Bertino and F. Marinaro. An evaluation of text access methods. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, 1989. Vol.II: Software Track*, 1989.
2. A. K. Bhide, A. Dan, and D. M. Dias. A simple analysis of the LRU buffer policy and its relationship to buffer warm-up transient. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.
3. M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th VLDB Conference, Kyoto, Japan, August 1986*, 1986.
4. A. Eickler, C. A. Gerlhof, and D. Kossmann. Performance evaluation of OID mapping techniques. In *Proceedings of the 21st VLDB Conference*, 1995.
5. R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic B^+-tree. In A. U. Tansel, J. Clifford, S. K. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal databases: theory, design and implementation.* The Benjamin/Cummings Publishing Company, Inc., 1993.
6. C. Faloutsos and R. Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *Proceedings of the 14th VLDB Conference*, 1988.
7. Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in OODBs. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.
8. D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD*, 1989.
9. K. Nørvåg and K. Bratbergsengen. Log-only temporal object storage. In *Proceedings of the 8th International Workshop on Database and Expert Systems Applications, DEXA'97*, 1997.
10. K. Nørvåg and K. Bratbergsengen. Optimizing OID indexing cost in temporal object-oriented database systems. In *Proceedings of the 5th International Conference on Foundations of Data Organization, FODO'98*, 1998.