Fine-Granularity Signature Caching in Object 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

In many of the emerging application areas for database systems, data is viewed as a collection of objects, the access pattern is navigational, and a large fraction of the accesses are perfect match accesses/queries on one or more words in text strings in the objects in these databases. A typical example of such an application area is XML/Web storage. In order to reduce the object access cost, *signature files* can be used. However, traditional signature file maintenance is costly, and to be beneficial, a low update rate and high query selectivity is needed to make the maintenance and use of signatures beneficial. In this report, we present the *SigCache* approach. Instead of storing the signatures in separate signature files, the signatures are stored together with their objects. In addition, the most frequently accessed signatures are stored in a main memory *signature cache* (SigCache). Because the signatures are much smaller than the objects, the increase in update cost is not significant, and the number of objects that actually have to be retrieved can be reduced to a fraction when using the signatures stored in the SigCache as a filter during perfect match accesses.
Keywords: object database systems, signature files, buffer management, query processing, XML storage

1 Introduction

In many of the emerging application areas for database systems, data is viewed as a collection of objects and the access pattern is navigational. A typical example of such an application area is XML/Web storage. XML data should preferably be stored in an object database system (ODB) [9], but it can also be stored in a relational database system or an object-relational database system. Another typical characteristic of the new applications is that a large fraction of the accesses are perfect match accesses on one or more words in text strings in the objects/tuples in these databases. For such accesses, *signature files*[1] can be used to reduce the query cost.

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

*IDI Technical Report 6/2000, ISSN 0802-6394.

[1] A signature is a bit string, which is generated by applying some hash function on some or all of the attributes of an object. Note that *signatures* are also often used in other contexts, e.g., function signatures and implementation signatures.

In this report, we present the *SigCache* approach. Instead of storing the signatures in separate signature files, the signatures are stored together with the objects, and the most frequently accessed signatures are in addition stored in a *signature cache* (SigCache). A signature is in general much smaller than an object, so that the number of signatures we can keep in the SigCache is much higher than the number of objects we can store in the main memory buffer. When an object is updated and the signature is stored on the same page, the extra insert cost is only marginal. The signatures can also be used to reduce the CPU cost when the objects are already in memory.

In this report, we describe in detail the use and maintenance of the SigCache, and analyze its performance by the use of cost functions. As for all access methods, the gain depends on access patterns. We show that the gain from using the SigCache approach is significant for most access patterns.

The discussion here is done in the context of ODBs, but the approach is also applicable for relational database systems and object-relational database systems experiencing a navigational access pattern. We assume a traditional client/server system, but it should also be noted that this approach could prove to be even more useful in the context of mobile databases, where reduction amount of data to be transferred between client and server has even more impact on performance than it has in traditional systems.

The organization of the rest of the report is as follows. In Section 2 we give an overview of related work. In Section 3 we give a brief introduction to signatures. In Section 4 we describe the SigCache approach. In Section 5 we develop a cost model, which we use in Section 6 to study the performance when a SigCache is used. Finally, in Section 7, we conclude the report and outline issues for further research.

2 Related work

Several studies have been done on using signatures as a text access method, e.g. [1, 3, 4, 7]. Less has been done in using signatures in ordinary query processing, but signature file techniques have been shown to be beneficial in queries on set-valued objects [5].

We have in a previous paper described how signatures can be used in ODBs that use logical OIDs [8]. In an ODB using logical OIDs, an OID index is used to map from logical OID to the physical location of the object, and the signatures can be stored together with the mapping information in this index. The problem with this approach, is that the OID index has to be updated for every object update, making it most suitable for relatively static data, or for temporal ODBs where the OID index is updated on every object update.

3 Signatures

In this section we describe how to generate signatures, how to use signatures to reduce the cost of perfect match accesses (PMA), and signature storage alternatives.

3.1 Signature generation

A signature can be generated by applying a hash function on some or all of the attributes of the object. By applying this hash function, we get a signature of F bits. If we denote the attributes of an object O_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

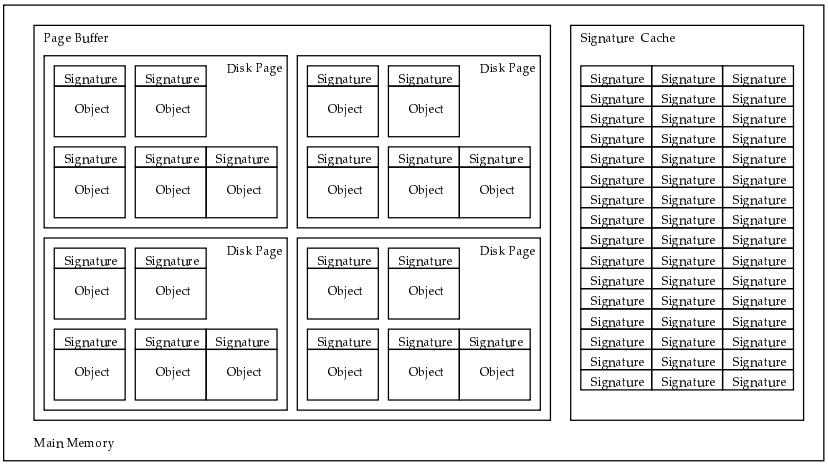

Figure 1: Storage of object pages and signatures in main memory. Page buffer to the left, and signature cache (SigCache) to the right.

generating function, and $A_j, \ldots A_k$ are some or all of the attributes of the object (not necessarily including all of $A_j, \ldots A_k$).

It is possible to generate the signature from the hash value of the concatenation of one or more of the attributes. However, such signatures can only be used for queries on the same set of attributes that were used to generate the signature. For this purpose, using an index will in many cases have a lower cost. In order 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 is generated for each attribute. The object signature is generated by performing a bitwise OR on each attribute signature. For example, 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 is very flexible in use. It can support several types of queries, with different attributes.

It is not necessary to use all attributes when creating the superimposed signature. If we know that only D of the attributes will be frequently used in PMAs, we can generate the signature from these attributes only. In the case of string attributes, for example in an XML object, we will generate a separate signature from each distinct word in the string attributes, and superimpose these word signatures. D will in this case be the number of distinct words in the object.

3.2 Using signatures

A typical example of the use of signatures, is a query to find all objects in a collection of objects where the attributes match a certain number of values, i.e., the query predicate is $Q = (A_j = v_j, \ldots, A_k = v_k)$. This can be done by calculating the query signature s_q of the query: $s_q = S_h(A_j = v_j, \ldots, A_k = v_k)$ (or $s_q = S_h(v_j)$ OR $S_h(v_i)$ OR $\ldots$ OR $S_h(v_k)$ if superimposed coding is used) 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*.

3.3 Signature files

Traditionally, the signatures have been stored in one or more signature files, separate from the objects/tuples. The files contain s_i for all objects O_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 cheaper 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 ODB, the pointer 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 [5]. Thus, BSSF based approaches are most appropriate for relatively static data.

To better support insertions, deletions, and updates, several dynamic signature file methods have been proposed. These are multiway tree variants and hash file variants.

4 The SigCache approach

An alternative to store the signatures in separate signature files is to store them together with the objects on the object pages, and in addition store the most frequently accessed signatures in a *signature cache* (SigCache). This is illustrated in Figure 1. The SigCache is a lookup table where replacement of signatures is done according to an LRU policy. In this report we assume that a clock algorithm with one access bit for each signature is used for this purpose.

A signature is stored on the same page as its object. This implies that if an object is resident in main memory (i.e., the page where the object resides is resident in the page buffer), its signature will also be resident, because it is stored in the same page. However, the opposite is not true: a signature can be resident in the SigCache even though its object is not resident in the buffer. When a page is discarded from the page buffer, signatures of some of the objects on the page can be resident in the SigCache.

Perfect match accesses can use the signatures to reduce the number of objects that have to be retrieved from disk. Only the candidate objects, with matching signatures, need to be retrieved. In order to identify a signature in the SigCache each signature need to have an unique identifier. In an ODB this will be the OID, in a relational- or object-relational database system this can be a physical tuple identifier or the concatenation of relation and key.

A signature is in general much smaller than an object, so that the number of signatures we can keep in the SigCache is much higher than the number of objects we can store in the main memory buffer. The fact that the effective space utilization in an object page buffer is low because of bad clustering on object pages further increase the amount of relevant signatures relative to relevant objects. The optimal size of the SigCache depends on access pattern and total buffer size relative to the total database

size, but a SigCache size in the order of 5-10% of the total buffer size will give a relatively stable performance (this will be discussed in more detail later) for a wide range of workload parameters.

Signatures are not maintained for all objects in the system, only when it is beneficial. This can for example be decided on the granularity of an object class or object container (also called file). Even when signatures exist, they are only used in a query if it is considered beneficial with respect to query cost. This is similar to traditional secondary indexes, where an index is created and maintained only if it is considered beneficial (this is decided by the database administrator), and the index is only used in a query if it is considered beneficial (this is decided during query planning).

4.1 The advantages of the SigCache approach

The SigCache approach has many advantages. The most important are:

- Traditionally, signatures have only been beneficial for relatively static data (i.e., a low update rate), for example text documents. Even when dynamic signature files are used, the update rate must be low if the use of signatures should improve performance. Dynamic signature files also have higher space requirements and search cost compared to SSF and BSSF.

  In contrast to using separate signature files, the SigCache approach is also useful in the case of high update rates. A signature is in general much smaller than the object it is created from, so that when an object is updated and the signature is stored on the same page, the extra insert cost is only marginal. If only a moderate amount of main memory is used for the SigCache, the page buffer hit rate is only marginally reduced.

- Read accesses in a relational database system are frequently set accesses which can benefit from traditional signature files. In contrast, read accesses in ODBs are mostly navigational. Even in the case of collection (for example a set) queries, navigation will often be the result. Unlike a relational database system where the queried set is a relation on storage, in an ODB, a collection can be a collection of references to objects (OIDs) rather than the objects themselves (an object can in this way belong to more than one collection). This navigation can make the average signature retrieval cost high if the signatures are stored in separate files. If the most frequently accessed signatures are stored in the SigCache, this cost can be significantly reduced.

- Previously, signatures have mostly been used to reduce the I/O-costs. However, signatures can also be used to reduce the CPU costs. Even if an object is resident in main memory, a signature comparison can be used before matching the attributes. Especially in the case of many or large attributes, this can reduce the CPU cost for a PMA.

4.2 Query processing using the SigCache

When a PMA is done on one or more attributes of an object, the following algorithm is used to determine if an object O_i is a match, and at the same time maintaining the contents of the SigCache:

1. A lookup is done for the object's signature s_i in the SigCache. If successful, the *access bit* of the signature in the SigCache is set.

   If this lookup is not successful, the signature has to be retrieved from the page where the object is stored. This page may be resident in the page buffer, but if it is not, the page has to be retrieved from disk. When the page is found, the signature s_i which is stored on the page, is inserted into the SigCache. The access bit for the signature is set when the signature is inserted.

2. The signature s_i is compared to the query signature s_q. If not all bit positions set to 1 in s_q are set to 1 in s_i, we know for sure that the object does not match the query predicate. If all bit positions set to 1 in s_q are set to 1 in s_i the object is a possible match, and we have to retrieve the object in order to compare the value of its attributes with the query predicate. The page where the object is stored on might already be in the page buffer (because it has been accessed recently, or has been brought into the buffer in order to retrieve the signature of one of the objects on the page), if not, the page has to be retrieved from disk.

A query on a collection of objects is done in the same way, once for each object. Also note that by employing signatures as outlined in the algorithm above, the CPU cost can also be reduced, because the signatures are compared before the object is compared with the search predicate.

4.3 Scan operations

One single scan operation can make all the signatures stored in the SigCache to be replaced. This is often not desired. One reason for this, is that the signatures retrieved during a scan operation will in general have less chance of being used again, it is not likely that the whole collection or container to be scanned represents a hot set. Even if this is the case, it is possible that the number of signatures retrieved during the scan is larger than the number of signatures that fits in the SigCache. In this case, even if we shortly after do a new scan over the collection/container, we will have a SigCache hit probability of 0. This is similar to general buffer management in the case of scan operations.

We have several possible strategies to use in order to avoid the problem with scan operations involving a large number of objects. One strategy is to not insert signatures retrieved in a scan operation into the SigCache (but the signatures already stored in the SigCache can be used when appropriate). If this strategy is used, we avoid the SigCache pollution, but at the same time we loose opportunities to benefit from the signatures, as many of these scan operations will involve a small number of objects. A variant of this strategy is to insert signatures into the SigCache if the scan operation involves a small or moderate number of objects, but not insert signatures if the scan involves a large number of objects.

It is often difficult to know the number of objects involved in a scan. It will probably be more safe to simply limit the number of signatures from objects of a certain class to be stored in the SigCache. For example, only 25% of the slots in the SigCache can be occupied by one class. This can be achieved by maintaining the number of signatures of objects from each object class in the SigCache. In many page servers the object class is not known to the server. In this case, the limitation can be the number of objects from a particular container/file (the container/file identifier is encoded into the OID in most ODBs).

4.4 Object updates and SigCache maintenance

Every time an object is modified, its signature has to be modified as well. A signature is stored together with its object, and with respect to concurrency control, logging and recovery, the signature is treated as a part of the object. If the signature of the object is resident in the SigCache, the signature in the SigCache has to be updated as well. This implies that a SigCache lookup has to be done for each object update. However, compared to the number of CPU instructions necessary to provide persistent storage of an object, this lookup cost is only marginal.

Only signatures that can contribute in read queries are beneficial to keep in the SigCache, so that when a signature is updated in the SigCache, the access bit is not set. Read accesses are necessary to make a signature stay in the SigCache.

4.5 Redundant signatures in main memory

A signature is stored in the same page as its object, so that if the object of a signature in the SigCache is resident in main memory, the signature is actually stored two places. In order to optimize memory usage, it is possible to only store signatures of objects that are not resident in main memory in the SigCache. This can be done by removing all signatures of objects in a page from the SigCache when the object page is retrieved into the page buffer. However, this is not a good strategy:

1. Removing and reinserting all signatures from a page significantly increases the CPU cost.
2. When a page is to be discarded from the page buffer, it is difficult to know which signatures should be reinserted into the SigCache (it is difficult to maintain the LRU policy). The only easy solution is to insert all the signatures of the objects on the page into the SigCache. However, in general only a few of the signatures on a page are "hot spot signatures", so this will pollute the SigCache. The result will be a serious reduction in the SigCache hit ratio, effectively killing all benefits from this memory "optimization".

The second problem in particular is very serious, so we do not consider this strategy any further.

4.6 Signatures and object-orientation

An object is not necessarily just a collection of simple value attributes as a tuple in a relational database system. An object can contain methods and references to other objects, and the objects in a queried collection can be objects of different classes, due to the concept of inheritance. We now describe how these issues can be handled with respect to signatures.

4.6.1 Methods

The use of signatures is quite straightforward in queries only involving value attributes. However, in an ODB it is also possible to access data from a general programming language, for example C++ or Java, and through method calls in queries (although not all vendors provide this functionality in their query languages).

Although the use of signatures is most beneficial in queries on value attributes, they can also be used in methods. One solution is to extend the equality operator in the programming language to compare the signatures before comparing the values. The advantage with this solution is that the use of signatures is transparent to the application programmer. Another solution is to explicitly use signatures by a separate function call before doing the comparison in the general programming language.

4.6.2 Complex objects and path expressions

When an object signature is generated, its attributes are simply treated as bit strings. If an attribute is an OID (a reference to another object), the OID is simply hashed like any other attribute.

Although not considered in this report, a value on a path expression could be used in the signature generation process, similar to path expression indexing. For example, it could be possible to define that instead of simply hashing a reference attribute `objPtr`, the value of $\texttt{objPtr}->\texttt{objPtr2}->\texttt{attrib}$ should be hashed instead. This would significantly increase the cost and complexity of signature creation, because every time `attrib` is updated, all signatures based on the value of `attrib` have to be updated as well. However, if queries on path expressions are common, this approach could still be beneficial.

4.6.3 Inheritance

Signatures for objects of two object classes A and B are *compatible*, i.e., can be compared, if 1) the attributes used for creating signatures for objects of one of the classes are a subset of the attributes used for creating signatures for objects of the other class, 2) the signature size is the same, and 3) the same hash function is used. This can be the case for signatures of a superclass and one of its subclasses.

Frequently, we want a larger signature size for a subclass when the number of attributes is higher. If this is the case, and we do a query on a collection of objects from the superclass as well as the subclass, a separate query signature has to be generated for each subclass (but note that only one signature for each object is generated and stored). When comparing the signature of an object with a query signature, the appropriate query signature is used, based on the class the object belongs to.

Example: Assume we have a class A, a class B that inherits from class A, and that we use a different signature sizes for class A and B. If we do a query over a collection of A objects, this collection can also contain B objects, because a B object is also an A object. When the query is executed, two query signatures are generated, one query signature s_q^A for objects of class A, and one query signature s_q^B for objects of class B.

4.7 Signature recalculation

It is not strictly necessary to store the signatures of the objects on disk. The signatures can be recalculated when the objects are retrieved. This saves disk space and increases the page buffer hit rate (because the total number of object pages is less), but increases the CPU cost. CPU speed is increasing at a much higher rate than disk performance, so even if signature recalculation instead of signature storage is not beneficial today, it is likely that this alternative will become attractive in the near future. In the main part of the analytical study in this report we will assume that the signatures are stored on disk, but we will also study how much the I/O costs can be reduced by using recalculation (Section 6.5).

4.8 Page vs. dual buffering

An alternative to fine granularity buffering of *signatures*, is to combine page buffering with fine granularity buffering of the *objects* instead. This is called *dual buffering*. If dual buffering is used, we try to keep well clustered pages in a page buffer, and objects from less clustered pages in an object buffer.

A thorough study of client side dual buffering by Kemper and Kossmann [6] showed that dual buffering can give a substantially higher buffer performance than a page buffer. However, the use of a dual buffer introduces several new options that makes tuning more complicated, for example when to copy an object from the page buffer to the object buffer, and when to copy a dirty object in the object buffer back to its home page. This makes it less clear how well dual buffering would perform in systems with complex workloads. The study showed that *dual buffering was most beneficial for a workload with read queries.* With update queries the gain was less or negative. Thus, we only consider the use of page buffering in this report. However, it should be noted that for workloads where dual buffering is beneficial, performance can still be improved by additional fine granularity buffering of signatures.

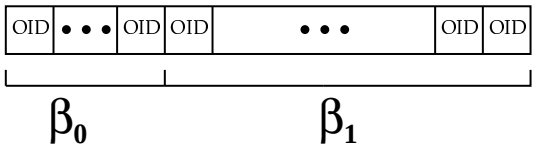

Figure 2: Partitioned data area. Each partition contains a fraction β_i of the data granules, and α_i of the accesses are done to each partition.

5 Analytical model

In order to study the gain from using signatures, we develop a cost model that models the object retrieval costs. The cost model focuses on disk access costs, as this is the most significant cost factor. We do not consider the additional update cost caused by storing the signatures, because the size of a signature compared to an object is small (between 3% and 6% in the study in this report). The purpose of this report is to analyze the effect of using signatures in an ODB, so we focus on navigational accesses and restrict this analysis to PMAs and signatures generated by the use of the superimposed coding technique.

The database system modeled in this report is a page server ODB. The ODB 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. The most recently used object pages are kept in a page buffer of size M_{pbuf}, and the most recently used signatures are kept in a SigCache of size M_{scache}. The main memory size M is the sum of the size of the page buffer and the SigCache, $M = M_{pbuf} + M_{scache}$.

5.1 Buffer hit rates

It is important to have an accurate buffer hit model. For this purpose, we use the Bhide, Dan and Dias LRU buffer model (BDD) [2]. An important feature of the BDD model, which makes it more powerful than some other models, is that it also can be used with *non-uniform access distributions.*

A database in the BDD model has a size of N data granules (e.g., pages), partitioned into p partitions. As illustrated in Figure 2, each partition contains a fraction β_i of the data granules, and α_i of the accesses are done to each partition. The distributions *within* each of the partitions are assumed to be uniform, and all accesses are assumed to be independent. We denote an access pattern (partitioning set) $\Pi = (\alpha_0, \ldots, \alpha_{p-1}, \beta_0, \ldots, \beta_{p-1})$. For example, for the 80/20 model, $\Pi_{2P8020} = (0.8, 0.2, 0.2, 0.8)$. In this report, we will study performance with four different access patterns:

Set	β_0	β_1	β_2	α_0	α_1	α_2
2P7030	0.30	0.70	-	0.70	0.30	-
2P8020	0.20	0.80	-	0.80	0.20	-
2P9010	0.10	0.90	-	0.90	0.10	-
2P9505	0.05	0.95	-	0.95	0.05	-

5.1.1 The BDD buffer model

We will briefly explain the main equations in the BDD model, the derivation and details behind the equations can be found in [2].

After n accesses to the database, the number of distinct data granules (pages, objects, or index entries) from partition i that have been accessed is:

$$N^i_{distinct}(n, N, \Pi) = \beta_i N(1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n})$$

The total number of distinct data granules accessed is:

$$N_{distinct}(n, N, \Pi) = \sum_{i=1}^{p} N^i_{distinct}(n, N, \Pi)$$

When the number of accesses n is such that the number of distinct data granules accessed is less than the buffer size B (the number of data granules that fits in the buffer), $\sum_{i=1}^{p} B_i(n) \leq B$, the buffer hit probability for partition i is:

$$P_i(n, \Pi) = 1 - (1 - \frac{1}{\beta_i N})^{\alpha_i n}$$

The overall buffer hit probability is:

$$P(n, \Pi) = \sum_{i=1}^{p} \alpha_i P_i(n, \Pi)$$

The steady state average buffer hit probability can be approximated to the buffer hit ratio when the buffer becomes full, i.e., n is chosen as the largest n that satisfies:

$$\sum_{i=1}^{p} N^i_{distinct}(n, N, \Pi) = B$$

We denote the average buffer hit probability as:

$$P_{buf}(B, N, \Pi) = P(n, \Pi)$$

where n is chosen as described above.

5.1.2 Page buffer hit rate

The number of disk pages necessary to store a database with N_{obj} objects, assuming page size S_P and an average object size of S_{obj} bytes:

$$N_{objpages} = \frac{N_{obj} S_{obj}}{S_P}$$

If we store F bits signatures of all objects on the object pages as well, the number of disk pages is:

$$N_{objpages} = \frac{N_{obj}(S_{obj} + \lceil F/8 \rceil)}{S_P}$$

With a page buffer size of M_{pbuf} bytes and an overhead for each item in the buffer of S_{oh} bytes, we can keep $N_{pbuf} = \frac{M_{pbuf}}{S_P + S_{oh}}$ of these pages in main memory. The buffer hit rate in this case is:

$$P_{buf\_page} = P_{buf}(N_{pbuf}, N_{objpages}, \Pi)$$

5.1.3 SigCache hit rate

For each signature in the SigCache, we need to store object identifying information in order to know which object it belongs to. It is not necessary to store the whole OID for each signature, variants of prefix compression or multi level access tables can be employed. Assuming S_{ID} bytes in average are needed to know which object a signature belongs to, the number of signatures that fits in the SigCache is:

$$N_{scache} = \frac{M_{scache}}{\lceil F/8 \rceil + S_{oh} + S_{ID}}$$

The SigCache hit rate is:

$$P_{scache} = P_{buf}(N_{scache}, N_{obj}, \Pi)$$

5.2 Object retrieval cost

One or more objects are stored on each disk page. If we denote the time to read or write a page as T_P, the average cost of reading one page from the database is:

$$T_{readpage} = (1 - P_{buf\_page})T_P$$

In order to reduce the object retrieval cost, objects are usually 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 $N_{o\_page} = N_{obj}/N_{objpages}$ objects on each page, and n of the objects on the page will be used before the page is discarded from the buffer, $C = \frac{n}{N_{o\_page}}$. If $N_{o\_page} < 1.0$, i.e., the average object size is larger than one disk page, we define $C = 1.0$. The average object retrieval cost is:

$$T^{nosig}_{readobj} = \frac{1}{CN_{o\_page}} T_{readpage}$$

We model the database read accesses as 1) *ordinary object accesses*, and 2) *perfect match accesses*, which can benefit from signatures. We assume the perfect match accesses to be a fraction P_{PMA} of the read accesses, and that P_A is the fraction of matched objects that are actual 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 $m = \frac{F \ln 2}{D}$. The average object retrieval cost employing signatures is:

$$\begin{aligned}
T^{sig}_{readobj} = &\text{ Cost of non-PMA access} \\
&\text{+Cost of PMA access where} \\
&\text{signature not in SigCache} \\
&\text{+Cost of PMA access where} \\
&\text{signature in SigCache} \\
= &(1 - P_{PMA})T^{nosig}_{readobj} \\
&+P_{PMA}(1 - P_{scache})T^{nosig}_{readobj} \\
&+P_{PMA}P_{scache}(P_A T^{nosig}_{readobj} \\
&\qquad +(1 - P_A)F_d T^{nosig}_{readobj})
\end{aligned}$$

This means that of the P_{PMA} accesses that are for perfect match, we only need to read the object page in the case of actual or false drops.

Parameter	Value
S_P	8 KB
T_P	7 ms
S_{oh}	8 bytes
S_{ID}	4 bytes

Table 1: Default system model parameters.

Parameter	Case I	Case II
S_{obj}	128 bytes	2048 bytes
C	0.2	0.8
N_{obj}	16 mill.	1 mill.
D	4 attributes	128 attributes
P_A	0.001	0.001
P_{PMA}	0.4	0.8
Π	2P9010	2P9010
F	32 bits	1024 bits

Table 2: Default workload parameters.

6 Performance

We have now derived the cost functions necessary to calculate the average object retrieval cost, 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.

Workload model In this analysis, we consider a database in a stable condition, with a total of N_{obj} objects. The system model parameters are summarized in Table 1. For the workload, we consider two main cases:

Case I: This is the workload of a traditional application. The object size is relatively small, and the signatures are generated from a small number of attributes. In such applications, there will be a mix of access types, and only some of them can benefit from using signatures.

Case II: This is the workload of one of the emerging application areas, for example XML storage. As described by DeWitt et al. [9], the space overhead would be very high if each XML element was stored as a separate object. Instead, one object is used to store one XML document, and the XML elements stored as "light-weight objects" inside one storage object. The result is larger objects than case I, and each object contains a higher number of attributes. The attributes are frequently text strings, and a large fraction of the queries in such systems will be for perfect match of one or more words. In order to be able to use signatures for such queries, the object signatures are generated by superimposing the individual text word signatures. As a result, the value of D will be large.

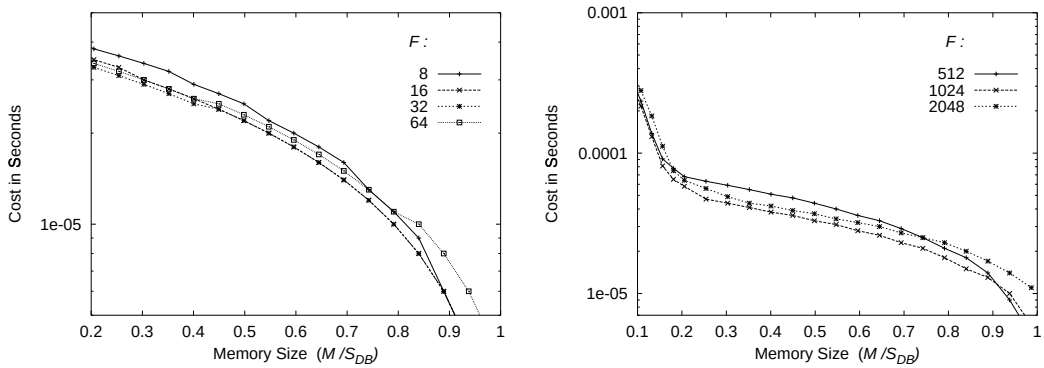

Figure 3: Cost when using different signature sizes. Case I to the left, and case II to the right.

The default parameter values for the two cases are given in Table 2. Note that with the default parameters, we keep the total database size constant, the studied database has a size of 2 GB for both case I and II. This is not a large database, but the interesting point is performance versus the relative memory size (memory size relative to database size). The results scale for larger database sizes, so that given a memory size of a fraction f of the total database size, the gain from using signatures is the same *independent of the database size*. On all figures, the memory size is given as memory size relative to the database size, i.e. as $\frac{M}{N_{obj}S_{obj}}$ (note that when signatures are used, the memory needed to keep all pages in memory is larger).

Also note that although the object retrieval cost depends on the clustering factor C, the gain of using signatures is independent of C, i.e., the gain is the same for different values of C.

6.1 Optimal signature size

Before we study the performance gain when using signatures, we have to determine appropriate signature sizes. The signature size is a tradeoff. Using large signatures reduces the false drop probability, but large signatures also reduces the number of signatures that can be kept in the SigCache. Too large signatures will also break the assumption that signatures are much smaller than the objects, and in that way increase the signature maintenance cost.

Figure 3 illustrates the access costs with different signature sizes. Note that in order to emphasize the interesting areas on the graphs, the cost scale is logarithmic and is cut. For case I, we see that $F = 32$ is a suitable signature size (for most systems this will also be the smallest reasonable signature size, because the CPU word size is 32 bits or larger). The signature size depends heavily on D, the number of attributes contributing to the signature. With an increasing value of D, the optimal signature size increases as well. This is clearly evident for case II, where $F = 1024$ is a suitable signature size.

When deciding the signature size it is also important to keep in mind that the signature size can not easily be changed afterwards if it is too small. If this should be done, reorganizing the database is necessary because there is not reserved space for larger signatures on the object pages.

6.2 Optimal SigCache size

The fraction of memory that should be used for caching signatures depends on the total memory size, database size, and workload. The SigCache size can either be static (but tunable), or it can be adaptive

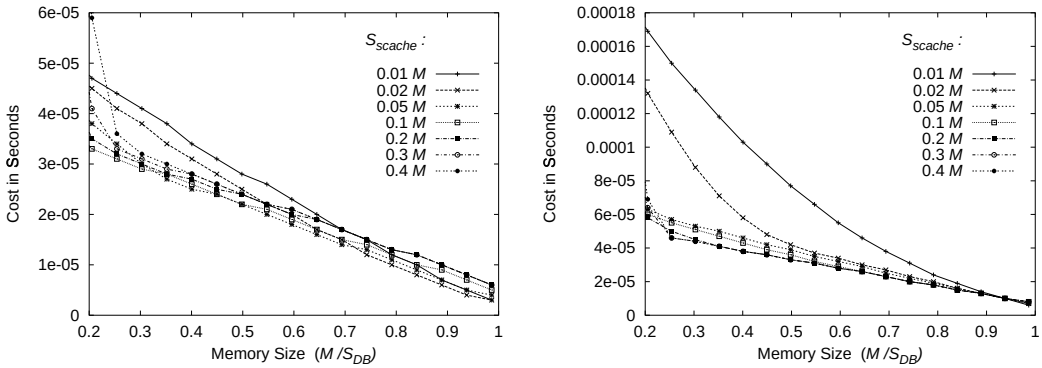

Figure 4: Cost with different amounts of memory reserved for signature caching. Case I to the left, and case II to the right.

(using cost functions to determine the size). It is often difficult to know the exact access pattern. In order to evaluate whether signatures are useful in practice, where the SigCache size can be far from optimal, we have studied how much performance is affected by different SigCache sizes. This is illustrated in Figure 4, which shows the average object access cost with different SigCache sizes.

Case I: The optimal fraction of memory used for caching signatures is between 0.05 and 0.1, depending on the total memory size. However, even if a larger size is chosen, the performance is still acceptable. The difference in cost is in this case not large. It should also be noted that keeping all signatures in the SigCache in this case requires 256 MB, so a SigCache larger than this size is of no use. Also, its is most important to keep the most frequently accesses signatures in the SigCache. When the SigCache is large enough to keep them, increasing the SigCache size further has less impact on performance.

Case II: In this case, the optimal fraction of memory used for caching signatures is in the order of 0.2, and is larger than for case I. The reason for this is the larger signature size used for case II. Given a certain SigCache size, the actual *number* of signatures that fits in the SigCache is smaller.

As can bee seen from the figure, the performance will also be acceptable with a fixed SigCache size. Even if using a size that is not optimal, good performance can still be achieved. For example, keeping the SigCache size constant at $0.1M$ gives good performance for a wide range of memory sizes and access patterns.

6.3 Gain from using signatures

Figure 5 shows the gain from using signatures, with different access patterns. The gain from using signatures is calculated as:

$$Gain = 100\Big(\frac{T^{nosig}_{readobj} - T^{sig}_{readobj}}{T^{sig}_{readobj}}\Big)$$

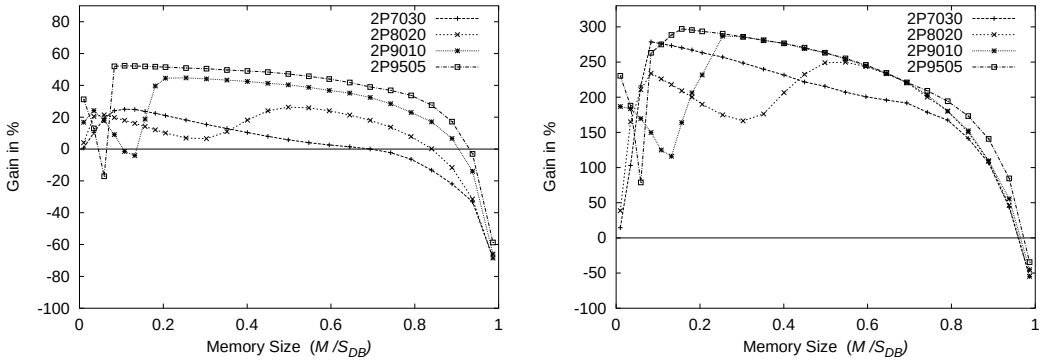

Figure 5: Gain from using signatures with different access patterns. Case I to the left, and case II to the right.

Case I: We see that using signatures is especially beneficial for access patterns with a narrow hot spot area. There are two cases when signatures are not beneficial:

1. When the memory size is large enough to keep most of the hot spot object pages. In this case, it is more beneficial to use all the memory for page buffering, so that the hot spot objects can be kept in main memory. It should be noted that when the memory size is smaller, so that only some of these pages fits in the page buffer, using signatures is beneficial.

2. The number of object pages necessary to store a database will be larger if signatures are stored as well. If the main memory is large enough to keep most of the object pages in main memory (large values of M) when signatures are not used, using signatures will result in decreased or negative gain because of a lower page buffer hit rate.

Case II: In this case, using signatures is very beneficial for all access patterns, except when most of the object pages fits in main memory, as explained above. The gain is high, up to 300% when we have a narrow hot spot area and medium amounts of memory.

6.4 The effect of P_A and P_{PMA}

The value of P_{PMA} is the fraction of the read accesses that can benefit from signatures. Figure 6 illustrates the gain with different values of P_{PMA}. As can be expected, a small value of P_{PMA} results in small or negative gain. As we increase the value of P_{PMA}, the gain increases. With large values of P_{PMA}, as we have assumed for case II, we can achieve a high gain. Interesting to note is that when using the same value of P_{PMA}, the gain for case I and case II is almost the same.

The value of P_A is the fraction of perfect match accesses that are actual drops (selectivity). The purpose of signatures is to reduce the number of objects that actually have to be retrieved, and we can only benefit from signatures if this number is sufficiently low. Figure 7 illustrates the gain with different values of P_A, and it is interesting to note that signatures will be beneficial even with a relatively large value of P_A.

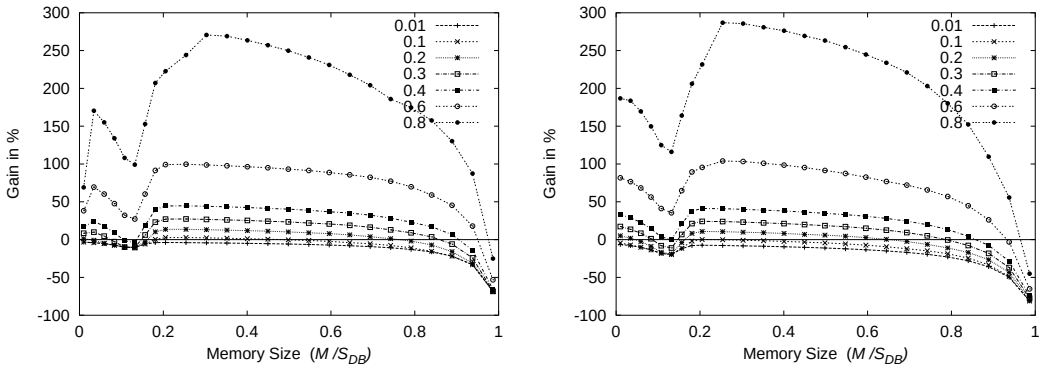

Figure 6: Gain from using signatures with different values of P_{PMA}. Case I to the left, and case II to the right.

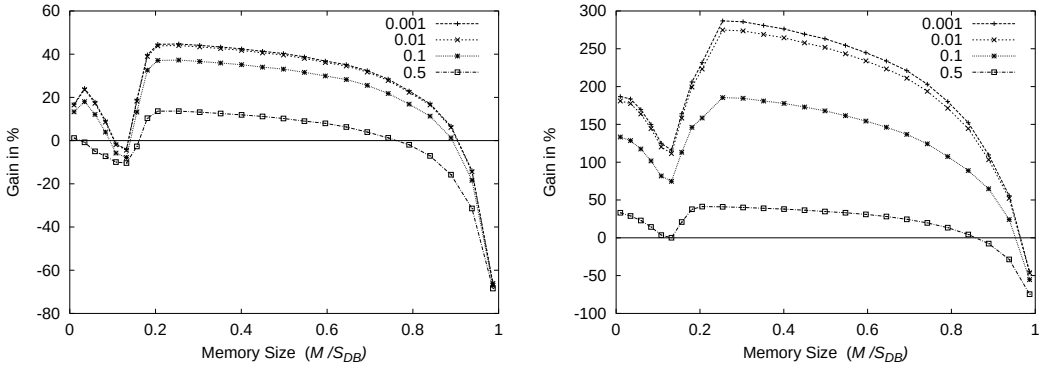

Figure 7: Gain from using signatures with different values of P_A. Case I to the left, and case II to the right.

6.5 The effect of recalculation

In the analysis so far, we have assumed that the signatures are stored on disk. However, as described in Section 4.7, it is also possible to recalculate the signatures when object pages are retrieved. With the current hardware, this will probably be too costly, but it is possible that it can be more interesting in the future. Figure 8 illustrates the difference in gain between recalculated signatures and storing signatures. The difference is large enough to make recalculation a possible strategy in the future when the CPU cost relative to I/O cost is even smaller than today.

7 Conclusions

We have described how object signatures can be cached in main memory in a signature cache, and how the signatures can be used to reduce the average object access cost in a database system. We developed a cost model that we used to analyze the performance of the proposed approaches, and this analysis showed that substantial gain can be achieved.

The example analyzed in this report is quite simple. Perfect match accesses have a low complexity,

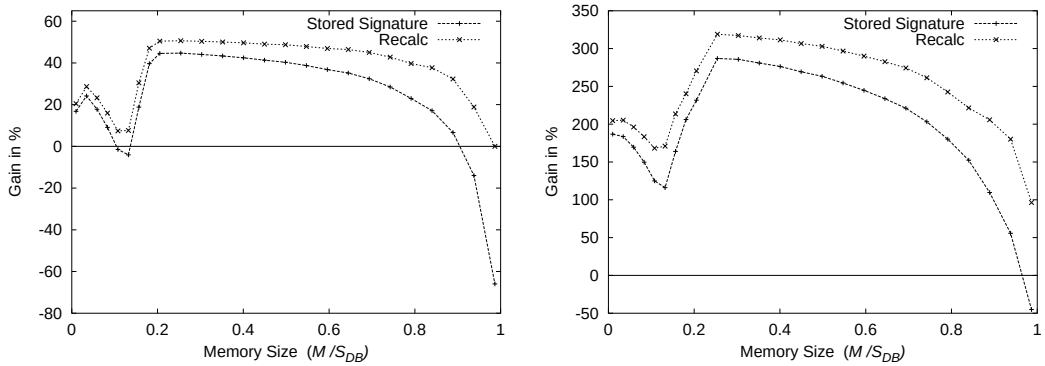

Figure 8: Gain from using signatures with recalculated signatures and stored signatures. Case I to the left, and case II to the right.

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.

Further work includes a study on how information about recent use of signatures of objects in a collection can be employed as part of the query planning. Recent use of the signatures implies a higher SigCache hit rate than would otherwise be expected, this knowledge can be used to enhance cost models used in query planning. Other further work include a study of using the cached signatures during join processing, which also involved perfect match accesses. Join is a frequent and costly operation, and the gain from using signatures and signature caching should increase performance considerably.

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] C. Faloutsos. Access methods for text. *ACM Computer Surveys*, 17(1), 1985.

[4] 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.

[5] 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.

[6] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. In *Proceedings of the 20th VLDB Conference*, 1994.

[7] D. L. Lee, Y. M. Kim, and G. Patel. Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3), 1995.

[8] K. Nørvåg. Efficient use of signatures in object-oriented database systems. In *Proceedings of Advances in Databases and Information Systems, ADBIS'99*, 1999.

[9] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies (submitted for publication), 2000.