CHAPTER 1

Chapter 1

Introduction

The advancements in the field of information technology during the past decades have revolutionized almost all data processing applications. This has entrusted researchers to design more powerful techniques to generate and manipulate large amounts of data to derive useful information. Indexing plays a vital role in the fast recovery of required data from large databases. Indexing that evolved with the concept of database management finds an extensive analysis and applications in the literature. Index structures facilitate the fast access to large sets of records by some key attribute. "Fast" means that the number of pages to be read to retrieve the qualifying records is small compared to the total number of pages the records occupy. Many indexing techniques have been developed and extensively investigated in both information retrieval and database research area. To index large files in information retrieval systems, different tree based indexing approaches have been proposed such as binary trees [KNUTH 73], B-tree [COMER 79], [BAYER 77] and its variants such as B+-tree and B*-tree [ELMASRI 89] etc. Most of these structures use the balancing mechanism of binary trees with some special features augmented to ease the tree balance or to minimize accesses to data files among which B+ tree is commonly preferred for its support to sequential and random access [LANGSAM 96]. Besides the direct access to records an indexing structure should also support different queries for records. The best general purpose index structure used for queries in many applications is the B+-tree as per the report of [KILGER 94].

An important question in information retrieval is how to create a database index which can be searched efficiently for the data one seeks. Today, one or more of the following three techniques have been frequently used: *full text searching, inversion* and the *signature file* [DERVOS 98], [CHEN 02]. Full text searching imposes no space overhead, but requires long response time. In contrast, inversion and the signature file work quickly, but need a large intermediary representation structure (index), which provides direct links to relevant data. The inverted index excels in query processing efficiency. It is a set of postings lists, each of which maps one keyword to a list of links to the data entries containing that keyword. Inverted indices can be implemented as sorted arrays, tries, B-trees and various hashing structures, whereby each real text block address (or document identifier) is stored more than once. The scheme needs to frequently undergo re-organization under intensive information insertion/updating procedures. If used in an object oriented database system (OODBS), the postings list of the inverted index will be a series of pairs of the form: (*C, oid*), where *C* represents a class name and *oid* represents an object identifier. In OODBS each object is assigned a system-wide unique object-identifier (OID), object instance is given an instance identifier (IID), which is the concatenation of a class identifier (CID) and an OID as studied by [KIM 90] and [SU 00] et al. Therefore, in the context of object-oriented databases, the inverted file will require much storage space.

Signature file discussed in [CHRISTO 84], [DESSMARK 98], [FALOUTSOS 85a] is a method quite different from the tree indexing techniques, by which each key word is assigned a signature and a set of key words is assigned a "super" signature constructed by superimposing the signatures in the set. It works as an inexact filter. The

signature file method which was originally introduced as a text indexing methodology [FALOUTSOS 85a], has now been applied to a wide range of applications, such as office filing, hypertext systems [FALOUTSOS 90] and relational and object-oriented databases [CHANG 89], [ISHIWAKA 93], [LEE 92], [YONG 94]. In comparison with the other index structures, it has mainly the following advantages:

- It can be used to efficiently evaluate set-oriented queries.
- It can handle insertion and update operations easily.

1.1 Signatures – A closer look

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,….,A_n, the signature of the object is $S_i = S_h(A_j,….A_k)$, where S_h is a hash value generating function, and $A_j,….A_k$ are some or all of the attributes of the object. The size of the signature is usually much smaller than the object itself [NORVAG 99].

1.1.1 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, $A_j = v_j,…..A_k = v_k$. This can be done by calculating the query signature s_q, of the query: $s_q = S_h(A_j = v_j,…..A_k = v_k)$. 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, called *actual 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 form 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*. The number of false drops can be statistically controlled by careful design of the signature extraction method [FALOUTSOS 87] and by using long signatures [FALOUTSOS 85a], [FALOUTSOS 87a].

1.1.2 Use of signatures

Compared to other index structures, signature file is more efficient in handling new insertions and queries on parts of words [CHEN 05], [CHEN 06] and especially suitable for set-oriented query evaluation.. Other advantages include its simple implementation and the ability to support a growing file. But it introduces information loss which can be minimized by carefully selecting the signature extraction method. More specifically, its output usually involves a number of false drops, which may be identified only by means of a full text scanning on every text block short-listed in the output. Also, for each query processed, the entire signature file needs to be searched [FALOUTSOS 85a]. Consequently, the signature file method involves high processing and I/O cost. This problem is mitigated by partitioning the signature file, by introducing auxiliary data structure, as well as by exploiting parallel computer architecture [CIACCIA 96].

1.1.3 Signature Generation

The methods used for generating the signature depend on the intended use of the signature. Some relevant methods are discussed below:

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 [ROBERTS 79], [DESSMARK 98]. 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 (i.e), 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.

1.1.4 Signature files

Intuitively, a signature file can be considered as a set of signatures. Signature files are based on the inexact filter. They provide a quick test, which discards many of the non qualifying elements. But the qualifying elements definitely pass the test although some elements which actually do not satisfy the search requirement may also pass it accidentally, i.e., there may exist "false hits" or "false drops" [FALOUTSOS 85a]. The signature of a text block or object can be obtained by superimposing (logical OR operation) all its constituent signatures (i.e) word signatures for text block and attributes' signatures for object. The set of all signatures form a signature file. To resolve a query, the query signature say s_q is generated using the same hash function and compared with signatures in the signature file for 1s sequentially and many non-qualifying objects are immediately rejected. If all the 1s of s_q matches with that of the signature in the file it is called a *drop*. The drop that actually matches the s_q is called an *actual drop* and drop that fails the test is called *false drop*. The next step in the query processing is the *false drop resolution*. To remove false drops each drop is accessed and checked individually. The number of false drops can be statistically controlled by careful design of the signature extraction method [FALOUTSOS 87] and by using long signatures [FALOUTSOS 85a], [FALOUTSOS 87a]. An example of superimposed coding for a text block of size 2 and a sample query evaluation is given below.

Information Retrieval	0010 0100 0100 0001
Block Signature	0110 0101

Sample queries

Matching query

Keyword = Information 0010 0100

Query descriptor 0010 0100

Block signature matches (Actual Drop)

False Match query

Keyword = Coding 0010 0001

Query descriptor 0010 0001

Block signature matches (False Drop)

but keyword does not

Non-matching query

Keyword = Information 0010 0100

Keyword = Science 0000 0110

Query descriptor 0010 0110

Block signature does not match

When a query arrives, the object signatures are scanned and many non qualifying objects are discarded. The rest are either checked (so that the "false drops" are discarded) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for attribute values. The query signature is then compared to every object signature in the signature file. Three possible outcomes of the comparison are exemplified below:

(i) The object matches the query; that is, for every bit set in s_q, the corresponding bit in the object signature s is also set (i.e., $s \wedge s_q = s_q$) and the object contains really the query word;

(ii) The object doesn't match the query (i.e., $s \wedge s_q \neq s_q$); and

(iii) The signature comparison indicates a match but the object in fact doesn't match the search criteria (false drop).

In order to eliminate false drops, the object must be examined after the object signature signifies a successful match. In addition, we can see that the signature matching is a kind of inexact matching. That is, s_q matches a signature s if for any bit set to 1 in s_q, the corresponding bit in s is also set to 1. However, for any bit set to 0 in s_q, it doesn't matter whether the corresponding bit in s is set to 1 or 0. The purpose of using a signature file is to screen out most of the non qualifying objects. A signature failing to match the query signature guarantees that the corresponding object can be ignored. Therefore, unnecessary object access is prevented. To determine the size of a signature file, the following formula is used [CHRISTO 84]:

$F \times \ln 2 = m \times D$, where F is the signature length,

m is the number of set bits and

D is the average size of a block

1.1.5 Research directions using signature files

Research on signature files can be roughly classified into two categories [LEE 95]. The *first* category focuses on new signature schemes for reducing the false drop probability without increasing the storage overheads. Numerous methods have been proposed and evaluated in the literature to reduce the number of false drops [FALOUTSOS 87], [FALOUTSOS 87a]. The *second* category of research is motivated by the fact that the search time on a signature file is directly proportional to the size of the text file, resulting in an unacceptable performance when the database is large. To alleviate the problem, many efficient search methods have been proposed in the literature, including the indexed descriptor file method [PFALTZ 80] and its variant S-tree [DEPPISCH 86] , the two-level superimposed coding method [CHANG 89] and the partitioned signature file method [LEE 89], [LEE 90]. These methods in general organize the signature file in a way such that only a small number of the signatures are accessed in response to a query.

Methods utilizing special hardware processors have also been proposed [LEE 86]. The research work reported in this thesis falls into the second category. It is prompted by the differences in performance measures and assumptions used by these various search methods. For instance, different coding schemes were used for generating the signatures. Disjoint coding was used in the indexed descriptor file for generating the block signatures while superimposed coding was used in other methods. Further, different techniques were employed to improve the search time (e.g., the partitioned signature file is based on hashing while the other methods are based on tree structures). The performance measures used in these methods were also different. The number of disk accesses was used in

some methods while signature reduction ratio was used in others. These wide differences make it difficult to compare the performance of different methods in a consistent manner.

1.2 Signatures in Object-Oriented Data Bases

Signature file techniques have been extensively investigated with respect to relational databases and text retrieval, and recently extended to OODBSs [ISHIWAKA 93], [LEE 92]. In Object-Oriented Data Bases (OODBs) each object is assigned a signature, and each class is assigned a signature file, which contains all the signatures of the objects belonging to it [CHEN 04]. In addition, the signature files have been proved to be useful in OODBs and efforts to integrate this technique into OODBs to improve the response time of a query is successful [YAN 94], [LEE 92]. OODBs have been designed to support diverse data types rather than just the simple tables, columns and rows of relational databases [ROBERT 01]. They can outperform relational databases at working with complex relationships among data [BAER 99]. In an OODB for instance, an object is represented by a set of attribute values. The signature of an attribute value is a hash-coded bit string of length F with m bits set to "1". Consider the example extracted from [CHEN 05] in which the signature of an attribute value "professor" can be constructed as follows. In terms of [CHRISTO 84], the letter triplets in a word (or an attribute value) are the best choice for information carrying text segment in the construction of the signature for that word. Then, we will decompose "professor" into a series of triplets: "pro," "rof," "ofe," "fes," "ess," and "sor." Using a hash function hash, we will map a triplet to an integer p indicating that the p^{th} bit in the string will be set to 1. Assume that we have hash (pro) = 2, hash (rof) = 4, hash (ofe) = 8, and hash (fes) = 9. Then, we will establish a bit string: 010 100 011 000 for "professor" as its word signature [DERVOS 98]. An object

signature is formed by superimposing the signatures for all its attribute values. Object signatures of a class will be stored sequentially in a file, called a signature file. Figure 1.1 depicts the signature generation and comparison process of an object having three attribute values: "John", "12345678", and "professor".

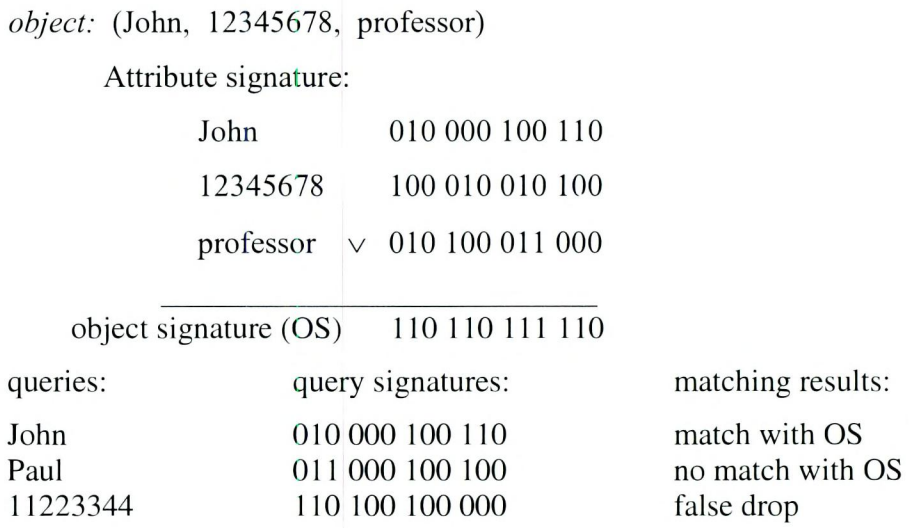

Figure 1.1 Signature generation and comparison

1.3 Need for the current work

Although several experimental and commercial systems, such as GemStone, Orion and O_2 have been developed, much work still needs to be done on query processing, optimization, and indexing techniques in order to improve performance. The powerful modeling capability is the major advantage of OODBSs over relational databases. Most OODBSs support secondary indexes on objects to improve retrieval speed. Most of the indexing techniques are based on tree or network structures. Key values in the index are linked with complex pointers in order to support complex query styles in OODBSs. The complexity results in large storage overhead and maintenance cost.

This thesis describes a new indexing technique for OODB using the dynamic balancing of B+ tree called Signature Declustering (SD) tree in which the positions of 1s in the signatures are distributed over a set of leaf nodes. Using this for a given query signature all the matching signatures can be retrieved cumulatively in a single node. Also for signature insertion and query searching an optimal search path is calculated so that the entire process is speeded up. Experiments have been conducted to analyze the time and space overhead of the SD-tree by varying the signature length and the distribution of signature weight. The study clearly indicates the advantage of fast retrieval time in a way quite different from the other methods suggested in the past.

In the next step we follow SD-tree based approach to object-oriented query handling. Query predicates in OODB are more complex due to their richer data model. *First* we apply and study the behavior of inclusive queries on SD (Signature Declustering) tree. It is observed that SD-tree which is retrieving all matching signatures in a single access handles both types of queries efficiently. Further the advantages of SD-tree like its flexibility and fast retrieval time is well-retained which promotes efficient handling of object-oriented queries. *Second* we focus on the object oriented query handling of nested queries in the class hierarchy using SD-tree. Our analysis shows that combined with query signature hierarchies the structure retrieves the matching objects quickly and therefore improves the time complexity of query evaluation substantially.

The chapters are organized as follows. Chapter 2 gives an introduction to the various signature file based techniques and their applications in OODB query processing.

The proposed SD-tree structure and its performance evaluation are discussed in Chapter 3. Applying SD-tree for inclusive queries in OODB is the content of Chapter 4. Chapter 5 studies the behavior of SD-tree on nested object query processing. Chapter 6 concludes the work and specifies the contribution to the field focussed with an outlook on future work.