Summary Database Index

Yuping Yang and Mukesh Singhal
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{yangy,singhal}@cis.ohio-state.edu

November 4, 1998

Abstract

We use signatures to construct access index in relational databases. Our method is different from previous methods in several ways. First, superimposing of signatures is done column-wise, not tuple-wise so that individuality of each attribute is kept in signatures. Second, the signatures are organized into a small relational database which we call the summary database. This allows a full set of relational operations, in particular join operations, to be carried out in the summary database. Third, access trees such as B+ trees are used as both search trees as well as mapping structures to map search result from the summary database to the original relational database.

For join operations, the summary database outperforms other existing join methods such as B_c tree, join index, and hybrid hash join for large relational databases with relatively large entries and queries with small query results, especially queries with conjunctive (more than two) joins and/or selections. For selection operations, the summary database is a light-weight alternative for index using the fully inverted file approach. The most significant performance advantage of the summary database is in executing conjunctive queries with multiple join and/or selection predicates.

The maintenance effort of a summary database is comparable to that of the signature file approach or access trees.

Key words: Database, query execution, index, summary, signature.

1 Introduction

Signatures as access indexes have been investigated extensively in the context of text databases [2, 3, 4, 9, 10, 14] and for other databases [11, 12, 13, 15]. Small index size, ease of maintenance, and simplicity are commonly recognized advantages of signature file indexing.

A signature is a 0/1 bit string hashed from a data item. Usually, for a particular application, all signatures have the same length, i.e., the number of bits. A data file can be partitioned and all signatures in a partition are superimposed (bitwise-ORed) to obtain a *partition signature*. A partition signature is an efficient means to store summary information for all data items in a partition. Suppose one of the data items in the partition is "OHIO" and the value "OHIO" is hashed to a signature of 300 bits of "0"s, except four bits, the b_1-th, b_2-th, b_3-th, and b_4-th bit positions have "1"s. Due to superimposing, the b_is bit position in the partition signature must also contain "1"s. For a query which wants to know whether "OHIO" is in the partition, the value "OHIO" is also hashed into a signature, query signature. All 300 bits of the query signature are "0"s except the four b_is positions have "1"s. If at every position of "1" bit in the query signature, the corresponding bit position in the partition signature also contains "1", then it is possible that the partition may contain the value "OHIO". Otherwise, the possibility that this partition contains value "OHIO" can be ruled out. By searching the signature file, a set of partitions, candidate partitions, can be found that potentially contain the query result. Only candidate partitions are further searched for the queried data item "OHIO". This practice can greatly reduce the query execution time since the partition signature file is usually much smaller (often 10% to 20% [14]) than the original data file and the union of all candidate partitions is also usually much smaller than the original data file.

A prominent feature of signature indexing is that a group of different data items can be indexed together by sharing a signature, and the grouping of data items does not have to satisfy any constraint. For a relational database, a data item is an entry at the cross point of a tuple (row) and an attribute (column).

Chang et al. [6] reported to use tuple signatures to embody summary information for tuples. A tuple signature is constructed by superimposing all signatures of individual entries of a tuple. The disadvantage of using tuple signatures is that the tuple signatures mix information from different attributes. There is no separation of information between different attributes. To use a file of tuple signatures as an access index for a relation is to treat the relation as a flat file and the two dimensional storage format of the relation is not utilized.

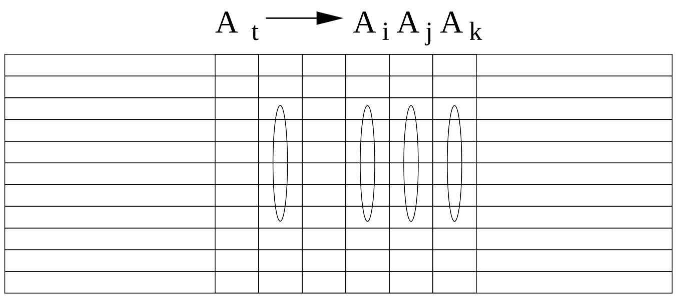

Figure 1: Grouping of entries in A_t induces grouping of entries in other attributes

A relation in a relational database is a two dimensional table and a partitioning of entries in one attribute can induce the partitioning of entries in all other attributes. This is illustrated in Fig.1.

In our proposed summary database index, we construct partition signatures by superimposing signatures from the same attribute (refer to Fig.1) instead of from the same tuple. This approach retains individuality of each attribute in signatures and enables query processor to execute relational operations, especially joins, in the signature file. This is a non-trivial departure from the previous methods in that the two dimensional tabular format of the relational database is reflected into the signature file and this format is utilized to speed up query executions for the database.

The mapping from the search results in the signature file to the original data file is an often neglected area in the signature file research. If the query result in the signature file can be precisely mapped to corresponding attributes and tuples in the original database, we can avoid wasting effort in loading large number of query-irrelevant disk pages into the memory for scanning. Many forms of access trees can be used for the mapping. We found that the B_c tree index proposed by Desai [7] is ideally suited for this purpose.

Section 2 proposes a new signature file indexing scheme, summary database. Section 3 proposes interval signature for indexing numerical data. Section 4 presents some partitioning algorithm and strategies for constructing small summary databases. Section 5 presents analyses and experiments to verify the filtering effect of signatures. Section 6 discusses the size and maintenance cost of the summary database. Section 7 compares the performance of summary database and B_c trees in relational databases. Section 8 concludes the paper.

2 Summary Database

In order to reap the biggest benefit of search space reduction in a database, we propose to organize signature files into a small relational database and call it *summary database*. A summary database includes summary relations, bonding tables, and mapping trees.

Summary relations: A summary relation contains partition signatures for a relation in the original database, the *main database*, based on partitions of values in a join attribute of the relation. A join attribute is an attribute in a relation that shares the same data domain with at least one more attribute in another relation. Let SJ be the set of all join attributes in the main database. SJ corresponds to a set of data domains SD. Every $D \in SD$ corresponds to at least two join attributes in SJ.

If a $D \in SD$ and a $J \in SJ$ is so related such that D is J's data domain, then the partitioning of values in D completely determines the partitioning of values in J.

The first step in constructing summary relations is to partition every $D \in SD$. There is a difference between a "data domain" and "all values in a data domain in a database". For convenience, we use the word "data domain" in the sense that it is the set of all values of some data domain in the database.

A partitioning of data domains induces a partitioning of join attributes which in turn induces a tuple partitioning in relations. In this way, all relations that have join attributes can be partitioned. We do not discuss the partitioning of relations without join attributes because in that case the relation can be treated separately (as an independent database as far as query execution is concerned)) from the rest of the database.

If a summary relation is constructed from a main database relation, the main database relation is called *base relation* of the summary relation.

More precisely, the algorithm to construct summary relations is as follows:

1. To partition every data domain $D_i \in SD$. For each data domain, an unique ID is assigned for each partition.

2. After each $D_i (\in SD)$ is partitioned, $D_i = \cup_{j=1}^{p_i} P_j$, each of D_i's corresponding join attributes J_1, ..., J_{k_i} can be partitioned as $J_t = \cup_{j=1}^{p_i} (J_t \cap P_j)$. In words, partitions in J_1, ..., J_{k_i} are formed by taking values from partitions in D_i. All partitions of J_1, ..., J_{k_i} that taken values from the same partition P_j in D_i is associated with the same ID of P_j.

3. Every partition in every join attribute induces a tuple partition in the relation of the join attribute. Signature is hashed out of every entry (not just entries of join attribute) of every tuple. All signatures belong to the same partition of the same attribute are superimposed (refer to Fig.1) to form a partition signature which is entered into a summary relation. Every entry in a summary relation contains exactly one partition signature and every tuple in a summary relation contains a tuple of partition signatures which correspond to a tuple partition (a set of tuples) in the base relation of the summary relation.

Bonding tables: A relation in the main database may have several join attributes. So, a relation in the main database may correspond to several summary relations as shown in Fig.2. The interconnecting relationship between summary relations is maintained through *bonding tables*. A bonding table acts in a similar way as a join index [5]. But a bonding table is between two summary relations instead of, as in the case of join index, between two relations in the main database.

A bonding table is a two dimensional tabular data table similar to a relation. Each column of a bonding table contains pointers to one of the summary relations connected by

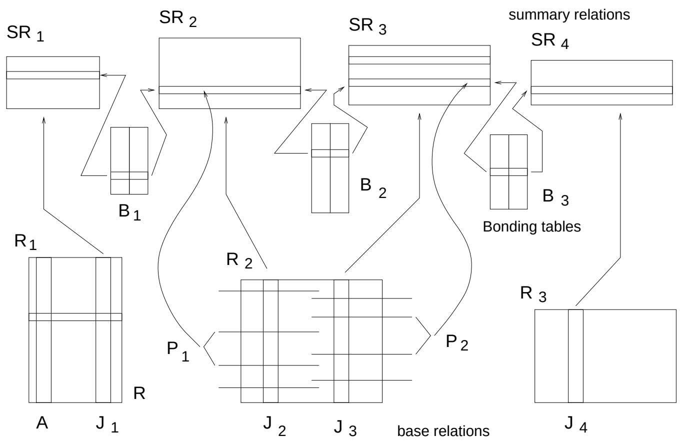

Figure 2: Summary relations and their bonding tables

the bonding table. The summary relations that a bonding table connects may have the same or different base relations in the main database.

For two summary relations which share the same base relation such as SR_2 and SR_3 in Fig.2, in general, there is no one-to-one correspondence between tuple partitions induced by the partitions of the two join attributes. Refer to Fig.2 and regard a tuple partition as a set of tuples, we consider a tuple partition P_1 (of J_2) corresponds to a tuple partition P_2 (of J_3) if and only if $P_1 \cap P_2 \neq \emptyset$. If P_1 corresponds to P_2, a row of two entries is established in the bonding table to connect P_1's *summary tuple* to P_2's *summary tuple*.

If two summary relations have different base relations as SR_1 and SR_2 in Fig.2, then, the tuple partitionings in the two base relations must be induced by the same partitioning of the same data domain. In this case, a row in the bonding table contains two pointers pointing to two summary tuples (in two summary relations) that obtained from the same data domain partition.

A bonding table with two columns is created between every pair of summary relations that share the same base relation. A bonding table is also created in every data domain which is the data domain of some join attributes and such bonding table may have two or more columns. The number of columns in a bonding table is equal to the number of join attributes that share the data domain.

The mapping of search results: After a query execution is completed in summary relations, a set of *summary tuples* has been found to be relevant to the query. These summary tuples correspond to some tuples, *base tuples*, in the main database and these base tuples constitute a super set of query results to the query.

The query results from summary relations can be mapped to the main database by associating pointers to summary tuples and these pointers point to every base tuples of the

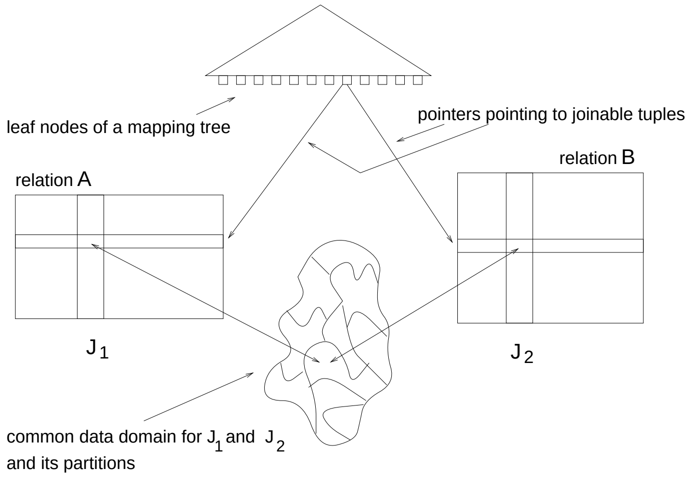

Figure 3: Mapping tree, data domain and join attributes

summary tuple. To improve access speed, these pointers can be organized into access trees. Since the partitioning of a join attribute is induced by a partitioning of its data domain, it is natural and efficient to create one mapping tree for each data domain of some join attribute and let the leaf nodes of the mapping tree contain pointers pointing to entries in different join attributes. This is shown in Fig.3. It is efficient because one data domain may correspond two or more join attributes. This mapping tree, if it is also a B+ tree, is actually a B_c tree which had been proposed by Desai [7].

Each leaf node of a B_c tree contains pointers pointing to all tuples in different base relations with the same join attribute value. So, besides serving as mapping trees for the summary database, B_c trees can also be used to execute relational joins. The join execution in B_c trees can further reduce the number of tuples need to be accessed for query execution in the main database.

It has been shown [7] that a B_c tree index, which contains many B_c trees, is a much smaller and faster access index for executing join operations in relational database than join index [5].

For every data domain of some join attributes, a B_c tree can be established as a mapping tree. A mapping using the B_c starts from the root of the tree and traverses down the tree until reaches some leaf nodes. Then, tuples of some base relations can be retrived following pointers in these leaf nodes.

We would like to mention that for some data domains of some join attributes, mapping trees may not be necessary. For example, two join attributes share a data domain (so they are joinable) and tuples in every tuple partition (induced by the partitions of the join attributes) in the two base relations of the two join attributes are physically stored together. In this case, a mapping tree is not necessary. Every summary tuple corresponds to a tuple partition. Since tuples in a tuple partition are physically stored together, only a starting address of the tuple partition is needed to access the base tuples of a summary tuple. However, join executions will be slower without B_c trees as mapping trees and more often than not, mapping trees are indeed necessary. For simplicity of discussion, in the rest of the paper we assume that a B_c is created for each data domain of some join attribute.

Query executions using a summary database: Joins, selections, as well as conjunctive queries (contain multiple join and/or selection predicates which should be satisfied simultaneously) can be executed in a summary database. The query execution result from the summary database is a set of tuples in the main database and these tuples constitute a super or exact set of tuples of the true query result for the main database.

For a relational database equipped with a summary database, a query is first executed in summary relations (possibly using bonding tables) and then executed in the leaf-nodes of B_c trees, and finally, relevant base tuples from the main database is loaded into the memory to finish off the query execution.

If a conjunctive query contains only (one or more) join predicates, the query execution can be completely finished in the leaf-nodes of B_c trees [7]. In this case, no loading of main database tuples is needed. If a conjunctive query contains at least one selection predicate, then the query result in the summary database is mapped to a super or exact set of the true query result (tuples) in the main database. These main database tuples are needed to be loaded in the memory to finish the query execution.

Refer to Fig.2, let R_1, R_2, and R_3 be three relations joinable through join attributes J_1 ($\in R_1$), J_2, J_3 ($\in R_2$), and J_4 ($\in R_3$); let SR_i denote the summary relation of J_i. Suppose a conjunctive query Q has three predicates:

$$
\begin{aligned}
Q \; &= Q_1 \;\; AND \;\; Q_2 \;\; AND \;\; Q_3, \\
& \qquad\qquad \begin{array}{llcl}
Q_1: & R_1.A & = & v, \\
Q_2: & R_1.J_1 & = & R_2.J_2, \\
Q_3: & R_2.J_3 & = & R_3.J_4.
\end{array}
\end{aligned}
$$

To execute this query in the summary database, v is hashed into a query signature $s(v)$ which is used to scan SR_1 to obtain a summary tuple set $s1$ in SR_1. Then, join Q_2 is executed between $s1$ and SR_2 by using bonding table B_1 to obtain a set of joined (concatenated) summary tuples $s2 \in SR_1 \bowtie SR_2$. Join Q_3 is executed between SR_3 and SR_4 by using bonding table B_3 to obtain a set of joined summary tuples $s3 \in SR_3 \bowtie SR_4$.

$s2$ and $s3$ can be further refined by using bonding table B_2 to obtain a set of joined summary tuples $s4 \in SR_3 \bowtie SR_4 \bowtie SR_3 \bowtie SR_4$.

Two B_c trees can map $s4$ to its base tuples: a B_c tree T_1 between J_1 and J_2 (they share a same data domain) and a B_c tree T_2 between J_3 and J_4 (they also share a same data domain).

$s4$ is mapped to a leaf node set $n2$ in T_1 and a leaf node set $n3$ in T_2. $n2$ and $n3$ are loaded into the memory and join executions are performed between them.

These join executions using leaf nodes further reduces the number of tuples that need to be loaded from the main database, and results in three tuple sets $t_1 \in R_1$, $t_2 \in R_2$, and $t_3 \in R_3$.

Finally, query execution is performed on tuple sets t_1, t_2, and t_3 to obtain query result for Q. Since t_1, t_2, and t_3 are much smaller than R_1, R_2, and R_3, query execution using summary database can be much faster than query execution performed directly on R_1, R_2, and R_3.

The size (in terms of bytes) of a typical leaf node is much smaller than a typical main database tuple. So, the additional overhead of executing a join using leaf nodes as described above is well justified by the saving in the query execution of main database tuples [7].

If the query Q does not contain selection predicate Q_1, then, the query execution in the main database can be completely avoided because a pure join operation can be executed by checking leaf nodes of the B_c tree [7].

The query execution in the summary database is very fast because of the small size of the summary relations and bonding tables, fast result mapping (from summary relations to the main database) by the B_c trees, and possibly (if join operations are involved) fast join executions using the leaf nodes of B_c trees [7]. The query result from the summary database greatly reduces the size of tuple set in the main database that is need to be considered for the query execution.

From above discussion, we can see that a database equipped with a summary database can execute queries very fast when the query result is small. When the query result is large, pruning using the summary database is not very effective. Actually, not only summary database, all indexed query execution techniques do not perform well when query results are large. So, in Section 7, we will analyze the query execution performance of a database equipped with a summary database under the assumption of small query result.

We emphasize that with or without joins, the query result obtained from a database equipped with a summary database is exact and complete.

3 Attributes with Numerical Data Domains:

Previously, signature methods are not used to index numerical data. This is because signatures are generated by hashing. Hashing does not preserve order among data. Numerical data are often involved in range queries and in range query order information is important. A query may look for all values in an attribute of a tabular dataset within an interval $[v_q - 1, v_q + 1]$. This small interval contains infinite number of values and clearly it is not feasible to have one signature for each value in this interval. We propose to use interval signatures to index numerical data and an example is used to illustrate our method.

A relation R has 200k tuples. The data domain of an attribute A_{i0} in R is an interval [0,300], which can be divided into 300 smaller intervals: [0,1], [1,2], ..., [299,300]. A hash function ϕ_{i0} is used to obtain interval signature S[i,i+1] from interval [i,i+1], $0 \leq$ i $<$ 300. For a numerical value $v \in$ [i,i+1], the interval signature S[i,i+1] is called the *interval signature of* v. Suppose the 200k tuples are partitioned into 1,000 partitions of approximately equal size. A partition signature is formed by superimposing all (about 200 of them) interval signatures of the data items in the partition. Note that the partitioning of data domain (interval) is independent of the partitioning of tuples in R.

Suppose a query asks for all data items in A_{i0} that fall within an interval $[v_q - 1, v_q + 1]$. If $v_q \in [i_q,i_q+1]$, then interval signatures of three intervals are used as query signatures: S[i_q-1,i_q], S[i_q,i_q+1], and S[i_q+1,i_q+2].

To answer this query, three query signatures are used to scan the signature file of 1,000 partition signatures. A partition whose partition signature is not matched with any query signature contains no data items in $[i_q - 1, i_q + 2]$ and will be excluded from further search and only those partitions whose partition signatures are matched are further searched.

The false drop ratio analysis of using interval signatures is the same as that for signatures obtained from text values and will be presented in Section 5.

4 Cascaded Partitioning Algorithm

Different tuple partitionings in the main database result in different degrees of interconnecting complexity between summary relations that share the same base relation. The size of bonding tables is affected by the interconnecting complexity between summary relations. In this section, we discuss partitioning strategies to reduce the size of bonding tables.

Let $J1$ and $J2$ be two joinable attributes of relation R. A partitioning of $J1$ results in a summary relation $S1$ and a partitioning of $J2$ results in a summary relation $S2$. $S1$ and $S2$ share the same base relation R. Suppose $J1$ is partitioned first. The partitions in $J1$ induces a tuple partitioning of R. The tuple partitioning induces a partitioning of $J2$. The partitioning of $J2$ may be further refined, i.e., partitioned, to make partitions more even. This practice results in a one-to-many relationship between tuples of $S1$ and tuples of $S2$. Under these partitions, the resulting bonding table linking $S1$ and $S2$ has no more rows than the number of tuples in $S2$.

To properly partition all join attributes in the database, we can rank all join attributes in the main database by following principles:

- P1: more important attributes are ranked higher. For example, in a supermarket database, price information may be ranked higher than color, shape, etc.
- P2: more frequently used attributes are ranked higher.
- P3: attributes with more orderly data domain are ranked higher. For example, the address of customers (which can be geographically grouped) is ranked higher than the name of the customers.
- P4: the join attribute in a larger relation is ranked higher. This is because it is more important for a larger relation to be evenly partitioned than a smaller one.

The cascaded partitioning algorithm can be briefly stated as follows. Join attribute with the highest rank should be partitioned first. To be more precise, the data domain of the highest ranking join attribute is partitioned first and this partitioning induces partitionings of values in all corresponding join attributes of the data domain. These partitionings in turn induce tuple partitions of the relations of the partitioned join attributes.

Join attributes are partitioned one by one in such a "cascaded" fashion. When the "cascaded" partitionings can not go further and there are still some join attributes not partitioned, the highest ranking join attribute among unpartitioned join attributes is picked and the "cascaded" partitioning resumes.

Eventually, all join attributes are partitioned and the algorithm stops. It is clear that bonding tables resulted from this partitioning algorithm do not contain many-to-many links. So, the number of tuples in a bonding table is limited by the maximum number of tuples of the summary relations that are connected by this bonding table.

These ranking principles are only heuristics and may be in conflict with each other when applied in practice. Should that happen, common sense and business knowledge should be applied to resolve those conflicts.

5 Filtering Effect of the Summary Relations

A false drop[1] is a match (see Section 1) of query signature with a partition signature but actually the queried data item is not in the partition. False drop ratio is the quotient of the number of false drops divided by the number of all drops, true and false. Long signatures have low false drop ratio but they take large space. Using the symbols listed in Table 1, the formula for computing false drop ratio is [1]:

$$F = (1 - (1 - k/b)^m)^k \tag{1}$$

[1]for historical reason, matches in signatures are called drops

n	number of data items, n = 125,000
m	partition size, $m = 50$
k	number of 1-bits in an individual signature, $k = 4$
b	length (number of bits) of a signature
q_j	number of join predicates in a conjunctive query
q_s	number of selection predicates in a conjunctive query
e	average number of bytes in a database entry, e=6
F	false drop ratio of using signatures

Table 1: Symbols used in size and false drop ratio analysis.

We give a brief explanation of the formula here. $1 - k/b$ is the probability that none of the k "1" bits of a signature is at a particular bit position. $(1 - k/b)^m$ is the similar probability for all m signatures in the partition. $1 - (1 - k/b)^m$ is the probability that at least one of the mk "1" bits of the m signatures is in the specified bit position. Finally, F is the probability that at least one "1" bit is in each of the k specified bit positions of the "1" bits of the query signature. We assume that a query signature is hashed from a single data item and has k "1" bits. F is actually the probability of match for an unsuccessful search. But it is also a good approximation for a successful search [2].

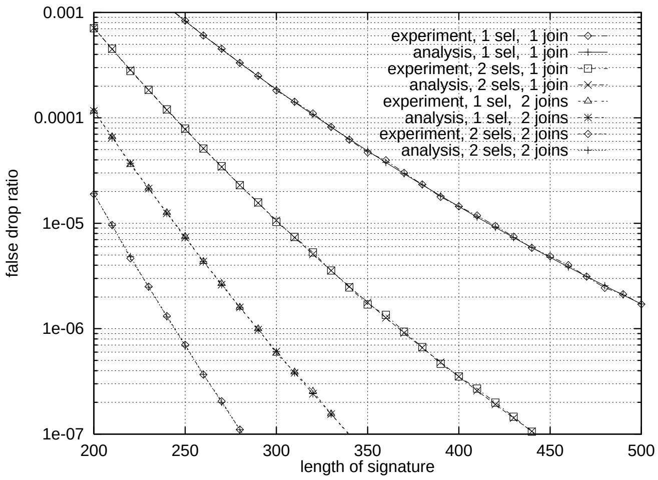

Figure 4: Signature lengths and false drop ratios.

We obtain the false drop ratio for a conjunctive query having q_j conjunctive join predicates (involving $2q_j$ summary relations) and q_s selection predicates (involving q_s summary relations) as:

$$F = ((1 - (1 - k/b)^m)^k)^{2q_j + q_s} \tag{2}$$

In Fig.4, the false drop ratios for different number of selection and join predicates in conjunctive queries are plotted using equation (2). We also conducted experiments using generated random values to verify these analytical results. Each plotted experimental result is the average of 30 runs with different seeds for the random number generator. In Fig.4, the experimental results are nearly identical to the analytical results.

The length of signatures directly determines the size of summary relations. When the length of signatures is 200 bits, the load factor is $200/50 = 4$, and when the length of signatures is 500 bits, the load factor is $500/50 = 10$. Refer to Fig.4 and Fig.5, we can see that with summary database size between 25% to 45% of the main database, very low false drop ratio can be achieved.

It is also clear that the use of summary relations is more effective in answering conjunctive queries having multiple selection and/or join predicates.

6 Size and Maintenance Cost

The size of a summary database can be estimated as follows. The size of all summary relations is about 5% - 20% of the main database. With the partitioning strategy introduced in Section 4, the size of all bonding tables can be made much smaller than the size of all summary relations. The size of all mapping trees has been analyzed in [7] and is also a few percents of the main database.

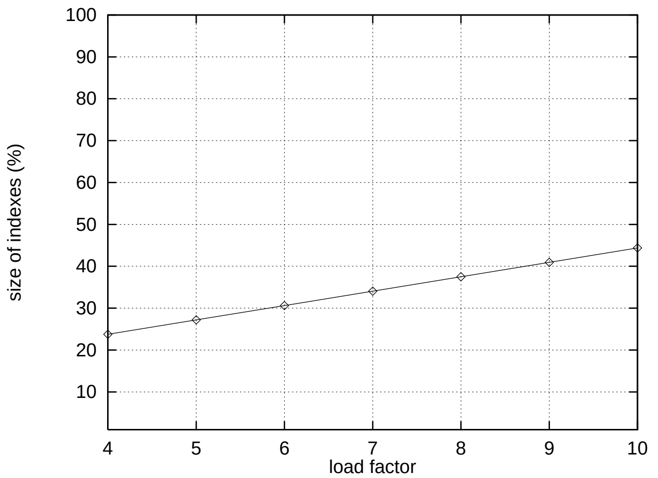

Figure 5: The relative size of a summary database

For selection operation alone, a summary database is capable of indexing all attributes in the main database. In comparison, the summary database as a selection index is much smaller than all access trees in a fully inverted file approach, i.e., the approach that to create one access tree per attribute. Besides, a summary database can also be used to execute joins and conjunctive queries.

Fig.5 shows the relative size of a summary database as compared to the main database under different load factor, i.e., the ratio of the total number of bits in the signature file divided by the total number of entries in the main database. The values of parameters used in the plotting are listed in Table 1. The size of all mapping trees is assumed to be 1/10 of the main database and the size of all bonding tables is assumed to be 1/10 the size of all summary relations. The size of index in a fully inverted file approach is not plotted because it depends on the size of the pointers in the index. Since every entry in the main database is pointed to by a pointer, the index size in a fully inverted index approach can be around or more than 100% of the size of the main database.

The maintenance cost of summary relations and bonding tables is comparable to any signature file approach [2, 4, 6, 9]. The maintenance cost of mapping trees, B_c trees, has been analyzed in [7]. In other words, there is no dramatic difference between the maintenance cost of the summary database and other indexes such as plain signature files and B_c trees. So, the maintenance cost of a summary database is acceptable in practice.

We emphasize that the construction of summary relations, bonding tables, and mapping trees is not done at the time of a query execution. As any index, once a summary database has been built, any number of queries can be executed without incurring any additional cost of updating as long as values in the main database are not updated.

7 Performance Comparison

B_c tree method [7] has faster join execution than join index [5], which in turn is faster than other more conventional join methods such as hybrid hash join and materialized view [8] when database is large and join selectivity is low. Our summary database is also suited for use in large databases and for executing queries with typically small query results. So, we only compare the performance between B_c tree method and our summary database index. We use the following query for performance comparison:

$$
\begin{aligned}
Q &= Q_1 \quad AND \quad Q_2, \\
&\quad Q_1: \quad R_1.A \quad = \quad v, \\
&\quad Q_2: \quad R_1.J_1 \quad = \quad R_2.J_2
\end{aligned}
$$

Assumptions: Relations R_1 and R_2 are assumed to have the same average tuple size (= number of attributes $\times$ average size of entries). On the average, each disk page contains 5 tuples. All other parameters are listed in Table 2. The cost of loading joinable tuple from both relations and the cost of writing query results are not included in the comparison since

$P_{sz} = 2,048$	page size (in bytes) of disk pages
$\mid R_1 \mid$=200k	# pages of R_1, $\mid R_1 \mid = \mid R_2 \mid$
$\mid\mid R_1 \mid\mid$	# tuples of R_1 ($= 5* \mid R_1 \mid$)
α	proportion of joinable tuples in a relation, $\alpha = 0.1\%$
$m = 50$	partition size (for constructing the summary database)
$TID_{sz} = 4$	the size (in bytes) of a TID
$A = 10$	the average size (in bytes) of data entries
b	the length (bits) of signatures, b = 200, ..., 500

Table 2: Parameters used in the performance comparison.

it is the same for both methods. The cost to write out query results are omitted for both methods since it is the same for both methods.

Cost of a query execution by using B_c trees: The selection (predicate Q_1) costs $\mid R_1 \mid$ sequential disk I/O and for the join execution, leaf nodes of a B_c are loaded into the memory. So, the query execution cost is roughly:

$$C_{B_c} = \mid R_1 \mid + (\mid\mid R_1 \mid\mid + \mid\mid R_2 \mid\mid) * (A + TID_{sz})/P_{sz} \tag{3}$$

The first term is cost of selection and the second term is the cost to load all leaf nodes of a B_c tree into the memory for join execution.

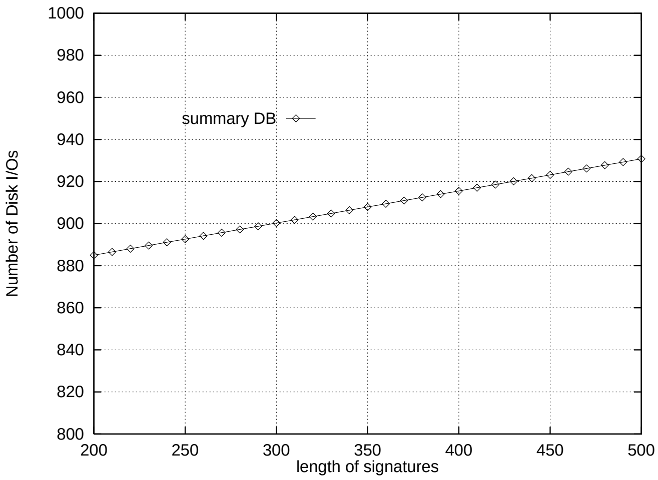

Figure 6: Query execution cost for using summary database

Cost of a query execution by using a summary database: As discussed in Section 5, false drop ratio can be made very low by using relatively long signatures. To make the cost estimation simple, we assume the false drop ratio is zero. Another simplification we made is to neglect the pruning effect of using summary tuples to do join execution. This pruning effect allows less leaf nodes of a B_c tree to be loaded. The first simplification slightly favors the summary database method and the second simplification slightly favors the B_c tree method. The relative performance between these two methods also depends on the length of signatures and the size of partition. In the the comparison, the length of signatures is varied from 200 bits to 500 bits and the size of partition is fixed at 50.

When the query result is small, the selection (predicate Q_1) does not add any additional execution cost since the selection can be done in retrieved matching tuples. So, the cost of the query execution is:

$$\begin{aligned} C_{summary)} = \quad & 2 * (b* \mid\mid R_1 \mid\mid /m)/(8 * P_{sz})+ \\ & (\mid\mid R_1 \mid\mid + \mid\mid R_2 \mid\mid) * (A + TID_{sz})/P_{sz} \end{aligned} \tag{4}$$

The factor 8 in the first term is to convert bits into bytes and the whole first term is the number of pages of the summary relation of R_1 and R_2. The second term is the same as that in the B_c tree method.

Using equation (3) and parameter values in Table 2, the cost of query execution in the B_c tree method is 13355 disk I/Os. We varied the length b of signatures and the cost of query execution in the summary database method is plotted in Fig.6, which is much lower than that in the B_c tree method.

We should point out that when the conjunctive query has more predicates, the relative performance advantage of the summary database index over the B_c tree method will be more prominent. This is because when using the summary database index, the number of joined (concatenated from tuples in several relations) tuples that simultaneously satisfy several join and/or selection predicates will be even less, and for each mapping tree (in the summary database) involved, the number of leaf nodes that need to be loaded will be much less than the full B_c tree.

The summary database index can also be used for executing conjunctive queries with only (one or more) selection predicates. Fig.7 shows the comparison to execute a conjunctive query with two or four selection predicates between the summary database and the approach to use two or four B+ trees. Both selection predicates selects data items in the same relation. The false drop ratio is varied from 0.1 to 0.2 and other parameters are listed in Table 1. Fig.7 shows that when the false drop ratio is low and selectivity is also low, the summary database outperforms the approach to use multiple B+ trees to execute queries with two or more conjunctive selection predicates. An interesting feature of summary database is that the cost to execute a conjunctive selection query does not increase when the number of selection predicates increased. This is in sharply constrast to the approach of using multiple B+ trees: the more selection predicates in the query, the more B+ trees are accessed.

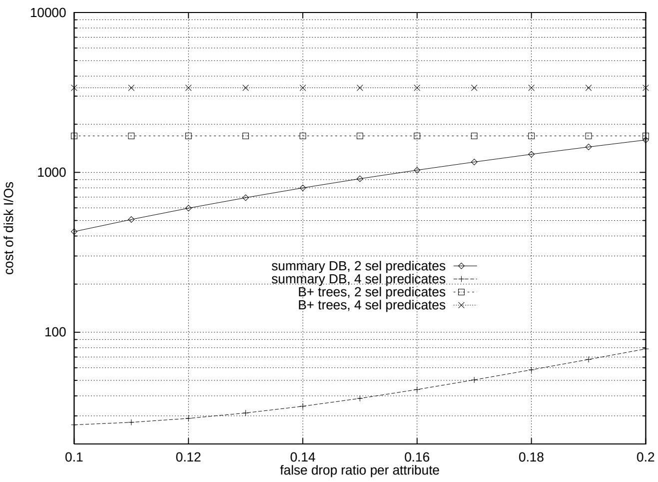

Figure 7: Query execution costs between summary database and B+ trees

8 Summary Remarks

The novelty of our method is that a small yet complete database, summary database, that contains only summary information of a relational database and has a similar schema as the relational database, is used to as an access index to facilitate fast join, selection, and conjunctive query executions in the relational database.

For join executions in large databases, the summary database index is faster than the B_c tree method, which in turn is faster [7] than the join index method [5]. Join index has been shown to have better performance than hybrid hash join and materialized view in large databases [8] with low join selectivities. So, the summary database index has better join execution performance than a variety of join methods in very large databases with low join selectivities.

A summary database index provides a fast access for each attribute in a relational database. Compared to the large storage space of a fully inverted file for each relation, a summary database is a light-weight access index which can provide comparable performance for selection executions.

The most significant performance advantage of a summary database is in the execution of conjunctive queries, i.e., queries with multiple join and/or selection predicates that connected by logical “AND”.

As with any indexing scheme, a summary database involves maintenance cost. However, this cost is comparable, or even lower than the cost to maintain a fully inverted file for every relation in the main database.

Future databases are likely to be much larger than today’s databases, and there is an increasing demand for various “online” service databases, such as databases employed over the World Wide Web or in mobile computing environments. Queries for these online databases typically have very low selectivities. The excellent performance in answering queries with low selectivity queries in very large databases makes the summary database index very attractive for such applications.

References

[1] R. Sacks-Davis, K. Ramamohanarao. "A two level superimposed coding scheme for partial match retrieval," *Information Systems, 8, 4, pp. 273-280, 1983.*

[2] C. Faloutsos, S. Christodoulakis, "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," *ACM Transactions on Office Information Systems, Vol. 2, No. 4, October 1984, pp. 267-288.*

[3] U. Deppisch, "S-Tree: A Dynamic Balanced Signature Index for Office Retrieval," *Proc of the ACM Conference on Research and Development in Information Retrieval, Pisa, Italy, Sept. 1986, pp. 77-87.*

[4] C. Faloutsos, S. Christodoulakis, "Description and Performance analysis of signature file methods for office filing," *ACM Transactions on Office Information Systems, Vol. 5, No. 3, July 1987, pp. 237-257.*

[5] P. Valduriez. "Join Indices," *ACM Transactions on Database Systems, Vol.12, No.2, pp. 218-246, June 1987.*

[6] W. W. Chang, H. J. Schek. "A Signature Access Method for the Starburst Database System," Proc. of the 15th Int'l Conf. on Very Large Data Bases, pp. 145-153, 1989.

[7] B. C. Desai. "Performance of a Composite Attribute and Join Index". *IEEE Transactions on Software Engineering, Vol.15, No.2, pp. 142-152, February, 1989.*

[8] J. A. Blakeley, and N. L. Martin. "Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis". *Proc of 6th Int'l Conf. on Data Engineering, IEEE, pp. 256-263, Feb 5-9, 1990.*

[9] C. Faloutsos, "Signature-based text retrieval methods: A survey," *Data Engineering Bulletin, Vol. 13, No. 1, March 1990, pp. 25-32.*

[10] D.L. Lee, C.-W. Leng, "A partitioned signature file structure for multiattribute and text retrieval," *Proceedings of the 6th Int'l Conf. on Data Engineering, Los Angeles, Feb. 1990, pp. 389-397.*

[11] W.-C. Lee and D.L. Lee, "Signature File Methods for Indexing Object-Oriented Database Systems." *Proc. 2nd Int'l Computer Science Conference,* Hong Kong, Dec. 1992, pp. 616-622.

[12] Y. Ishikawa, H. Kitagawa, N. Ohbo, "Evaluation of Signature Files as Set Access Facilitates in OODBs," *Proc. 1993 SIGMOD Conference,* Washingont, DC, June 1993, pp. 247-256.

[13] H.-S. Yong, S. Lee, H.-J Kim, "Applying signatures for forward traversal query processing in object-oriented databases," *Proc. 1993 SIGMOD Conference,* Washington, DC, 1993, pp. 518-525.

[14] D. L. Lee, Y. M. Kim, G. Patel. "Efficient Signature File Methods for Text Retrieval," *IEEE Transactions on Data and Knowledge Engineering, Vol. 7, No. 3, pp.357-367, June 1995.*

[15] D.L. Lee and W.-C. Lee, "Signature Path Dictionary for Nested Object Query Processing." *Processings of the IEEE International Phoenix Conference on Computers and Communications* (IPCCC '96), March 27-29, 1996, pp. 275-281.