Summary Databases as Indexing Structures

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

July 31, 1998

Abstract

A new summary database (SDB) method is proposed for fast query executions in large relational databases. We use signatures to embody summary information. The join execution performance of the SDB method is better than that of the B_c tree method [7]. Query executions using the SDB method is especially fast in very large databases for queries with low join selectivity or complex queries having multiple predicates. A distinct feature of the SDB method is that to process query in a database, a full set of relational operations, including joins and/or selections, can be executed in a small SDB before the large relational database is accessed. A detailed performance comparison between the SDB and B_c tree methods is provided. The excellent performance in answering queries with low selectivity joins and complex queries in very large relational databases makes the SDB method ideal for a variety of "online" service databases and data warehouses.

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

1 Introduction

Today, most large, commercially used databases are relational databases. With its mathematical soundness and relative simplicity, relational databases will continue to dominate the database industry in the foreseeable future.

Join execution is one of the most expensive operations in the query processing in relational databases because it typically requires several passes over entire relations. Techniques for executing joins can be classified as: (1) non-indexed: nested loop join, sort-merge join, and hybrid hash join, etc.; (2) indexed: nested block join with B+ index, distributive join [11], join index [5], and B_c tree [7], etc.

If the size of a join result is small, then an indexed approach usually only takes a few random disk accesses and is much faster than sequentially scanning a entire relation. So, the environment of a large database with typically small join results (low join selectivity) favors the use of an indexed join method.

The simplest indexes for speeding up join executions are indexes that are originally built for performing selections such as the B+ tree. For clarity, we call these indexes *selection indexes*. A selection index can be used to speed up join executions along with nested looping (i.e., the nested block join with a B+ tree index), sorting [9], and hashing [11]. The join index [5] is one of the cross-relational indexes specially designed for join executions. It is not widely used [7] in actual implementation of relational database systems due largely to its excessive large disk storage requirement. The B_c tree [7] is another cross-relational index specially designed for join executions. It has a much smaller data structure than the join index method. Except for the join index and the B_c tree method, all other join methods require at least one pass of scanning over at least one of the joining relations. A scanning over a large relation is very time-consuming.

The layered database concept has been proposed mostly for improving the modeling capability of the database or database security [14, 20]. Some have used multiple layers as a way of incorporating various databases in a heterogeneous system such as the World Wide Web [18]. A multi-layer approach was also proposed to speed up the query execution [17]. However, the join execution cannot be made in the upper layer due to difficulties in executing joins in upper layers [17].

This paper proposes a new summary database (SDB) that can be used to speed up joins as well as selections in large relational databases. A schematic view of the structure of a SDB is shown in Fig.1. Components of a SDB are: summary relations, bonding tables, and addressing tables. Each summary relation is constructed out of a relation, the *base relation*, in the original relational database, which we call the *main database*. Bonding tables link summary relations that are constructed out of mutually joinable base relations in the main database. Addressing tables establish the connection between summary relations and base relations. A query is first executed in the summary relations with the assistance of bonding tables. The result of the query execution in the summary relations is mapped through addressing tables to base relations. Because summary relations are much smaller than their base relations, the query execution in the summary relations is much faster than that in the main database. The result of the join execution in the summary relations reduces the search space for query executions in the main database. Therefore, the query execution of the main database is much faster because of reduced search space.

Signature method has been proposed to be used in answering queries having selection predicates in relational databases [6]. In contrast, our SDB method can also answer queries containing join predicates. Another difference is that the SDB method uses a different

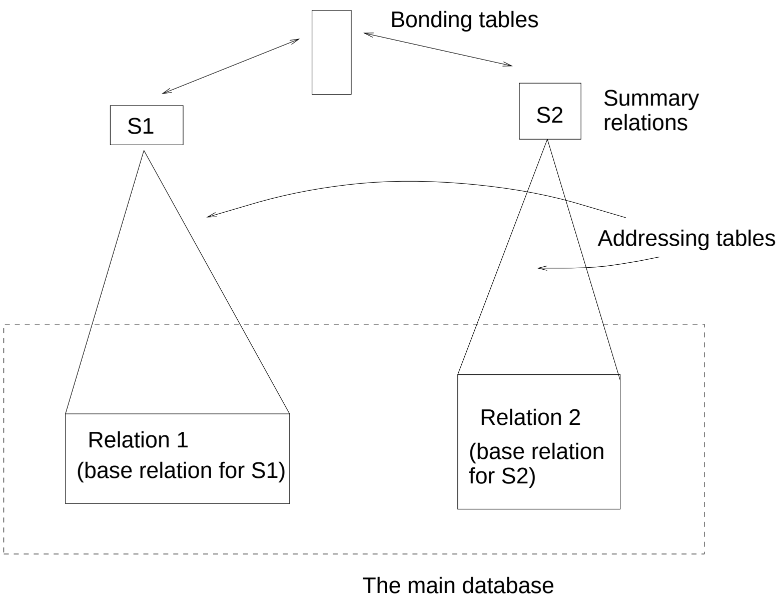

Figure 1: A sketch of a SDB and its original database.

signature extraction approach that will be elaborated later in this paper.

Our SDB method bears some resemblance to the join index method [5] and layered database approach [17]. The SDB method has join index like data structures, i.e., the bonding tables. However, bonding tables are over small summary relations in a SDB, not large relations in the main database. Also, through proper grouping of tuples in relations in the main database, the number of tuples in a bonding table can be controlled to be no more than the maximum number of tuples in summary relations that linked by the bonding table (see Section 3). Thus, we are able to overcome the problem of excessive disk storage space requirement as that in the join index [7]. By reducing the size of the data structure, the SDB method not only requires small disk space, but also delivers fast executing speed.

Our SDB method has closer resemblance to the B_c tree method than any other join execution method. The B_c tree method is the only join execution method that has join execution performance comparable to our SDB method when executing joins in very large relational databases with low join selectivities. Our SDB method outperforms the B_c tree method in join executions in very, very large databases. In other words, the larger the database, the relatively better the performance of the SDB method compared to that of the B_c method. In addition, our SDB method is not only a join execution method: it can execute joins as well as selections.

The paper is organized as follows. Section 2 introduces the proposed summary database structure. Section 3 presents a set of guidelines for grouping tuples in constructing the summary database. These guidelines are important because the way tuples are grouped critically affect the performance of the SDB method. Section 4 compares our SDB method with other join methods. Section 5 analyzes the size of the SDB and the pruning effect of the SDB. Analytical results are backed by simulation results. Section 6 gives a performance comparison between the SDB method and the B_c tree and the hybrid hash join methods. Finally, Section 7 concludes the paper.

2 The Summary Database (SDB)

The B_c tree method can execute joins faster than the join index because it uses a smaller data structure [7] and B+ trees. The summary database (SDB) method is a further improvement over the B_c method. In fact, the SDB method is more than just a join execution method. A summary database is an index structure such that multiple joins and/or selections can be executed in it.

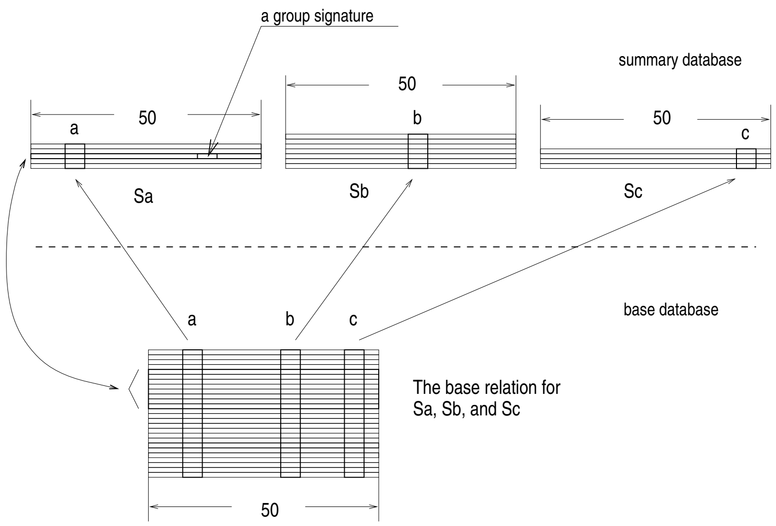

Figure 2: Correspondence between base relations and summary relations.

A SDB is constructed from a relational database (the *main database*). The SDB and its main database form a multi-layer database. For simplicity, we restrict our discussion to the case of only two layers: layer 0 is the main database and layer 1 contains the SDB. For a relation in the main database, if there are j join attributes[1], then j *summary relations* will be built in the SDB for that main database relation. The main database relation is called the *base* relation for these summary relations. Each summary relation has exactly the same schema, i.e., the same set of attributes, as its base relation. If a base relation has 50 attributes and three of them are join attributes, then there are three summary relations made from this base relation and each of the three summary relations has 50 attributes as

[1]A join attribute in a relation is an attribute that shares the same data domain with at least one more attribute in another relation, therefore a join between these two attributes is possible.

shown in Fig.2.

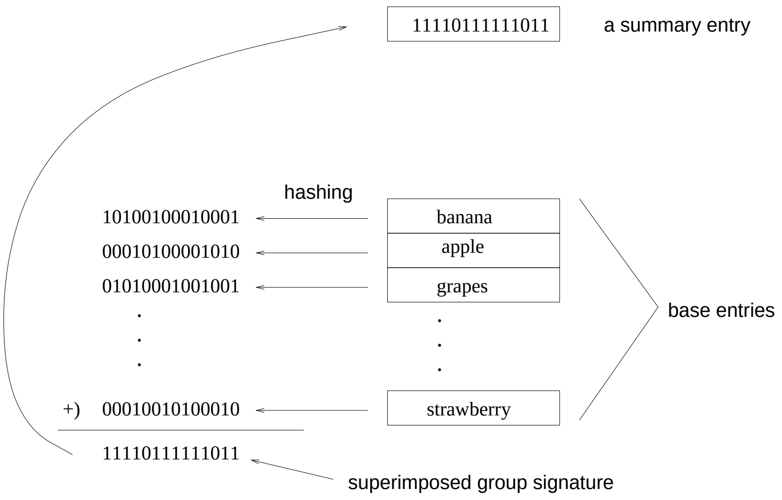

Figure 3: The making of an entry in a summary tuple.

A summary relation is made by grouping tuples and extracting signatures from its base relation. Each tuple (*summary tuple*) in a summary relation is made from a group (*base group*) of tuples (*base tuples*) in the base relation. As shown in Fig.3, each entry (which corresponds to an attribute) in a summary tuple contains a signature made from all entries of base tuples in the same attribute.

Fig.4 shows the overall structure of a SDB along with its main database.

Query execution starts from the SDB. Signatures are used to embody summary information (A description of the signature method and its application in the query execution is given in Appendix A). Query execution in a SDB is very fast, since a SDB is much smaller than the main database. After a query is executed in the SDB, the resulting summary tuples are mapped to their base tuples in the main database. The details of the query execution will become more clear after we have discussed bonding and addressing tables.

Construction of a SDB: Let A be a join attribute with a data domain D in a relation R. A shares data domain D with at least one more attribute in other relations of the database because A is a join attribute. Let S_D be the set of distinct values in all attributes in relations of the main database that have the same data domain D. Note that $S_D \subseteq D$. Values in S_D are partitioned (the partitioning algorithm is presented in Section 3) and each partition is given a unique identification number. Let us denote this unique identification number of each partition by PID. Note that PID represents a partition of values in a single data domain, not just a partition of values in a single attribute. A partition represented by a PID may contain values from several attributes.

From each partition in S_D, attribute A of R takes its own share of values, i.e., attribute

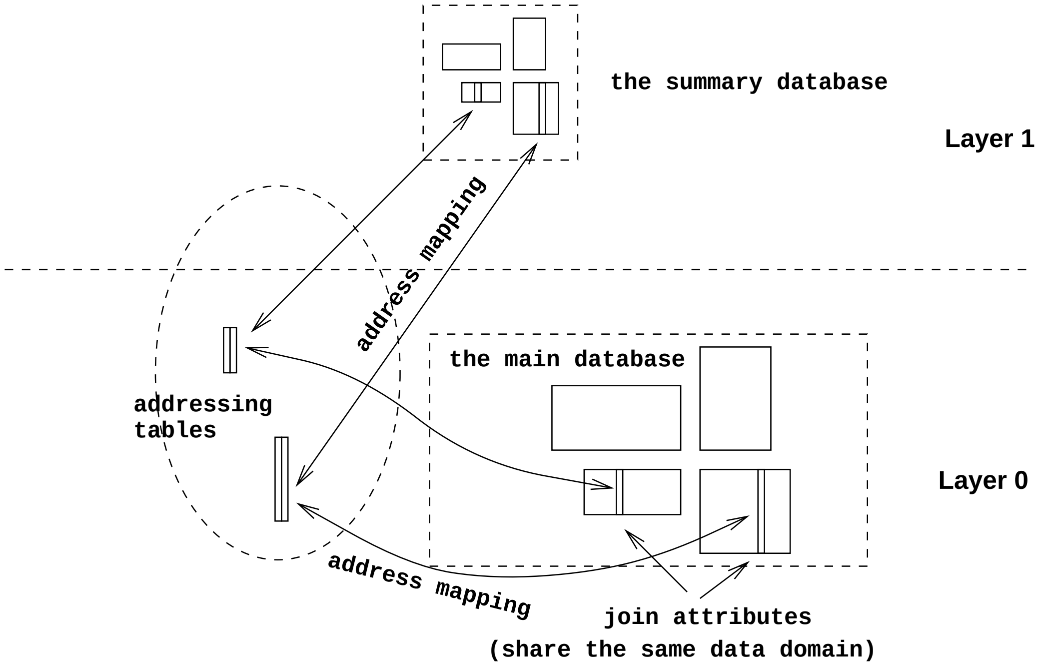

Figure 4: The architecture of a SDB along with its main database.

values in A, from the partition and these values form a group of values in attribute A. This group of attribute values determines a group of tuples in R. In this way the tuples in relation R are partitioned into groups.

A signature is hashed out of each entry of each tuple in a group (*the base group*). For each attribute (all attributes, not only join attributes. See Fig.2) in R and each group of tuples, a group signature is made by superimposing all signatures of this attribute in the base group. In this manner, a summary tuple is constructed from a group of base tuples in R. Each entry of the summary tuple contains a signature made from its base tuples. This summary tuple bears the same PID as the corresponding partition in S_D.

We emphasize that though the partitioning of tuples in a base relation is done based on the partitioning of values in join attributes, group signatures are made from all attributes in the base relation.

This process is repeated for every join attribute in the main database that has the same data domain. The number of summary relations constructed equals the number of attributes that share data domain D.

All summary tuples in different summary relations from the same partition of S_D bear the same PID. These PIDs logically link joinable summary tuples in different summary relations.

Note that the partitioning of S_D is done only once, and so is the hashing of values into individual signatures from values in a partition. The superimposing of individual signatures into group signatures is repeated for each attribute that have data domain D.

Let $S(A)$ be the set of all values in A. After S_D is partitioned, $S(A) \cap p$ is a partition in A if p is a partition of S_D. We say that the partitioning of S_D induced a partitioning of A.

Formally, let $P = \{p_1, p_2, ..., p_k\}$ be the set of all partitions of S_D (i.e., $S_D = \cup_{i=1,...,k}\, p_i$ and $p_i \cap p_j = \emptyset$ if $i \neq j$); let $J = \{j_1, j_2, ..., j_s\}$ be the set of all join attributes that have data domain D; let $S(j_t)$ be the set of values in j_t, $1 \leq t \leq s$ (Note that $\cup_{t=1,2,...,s}\, S(j_t) = S_D \subseteq D$). Then partitioning in each $S(j_t)$ $(1 \leq t \leq s)$ can be done as $PV = \{\ S(j_t) \cap p_1$, $S(j_t) \cap p_2$, ... , $S(j_t) \cap p_s\}$.

This unified partitioning strategy preserves connections between joinable summary tuples in different summary relations. However, it may cause skewed partitioning for values in some attributes that participate in this partitioning process. We will discuss this problem further and present its solution in Section 3.

Bonding tables: Bonding tables, as shown in Fig.5, are constructed between joinable summary relations with different base relations and between every pair of summary relations of the same base relation. A bonding table between joinable summary relations with different base relations is used to execute joins between summary relations. A bonding table between summary relations of the same base relation is used to propagate query result from one summary relation to another.

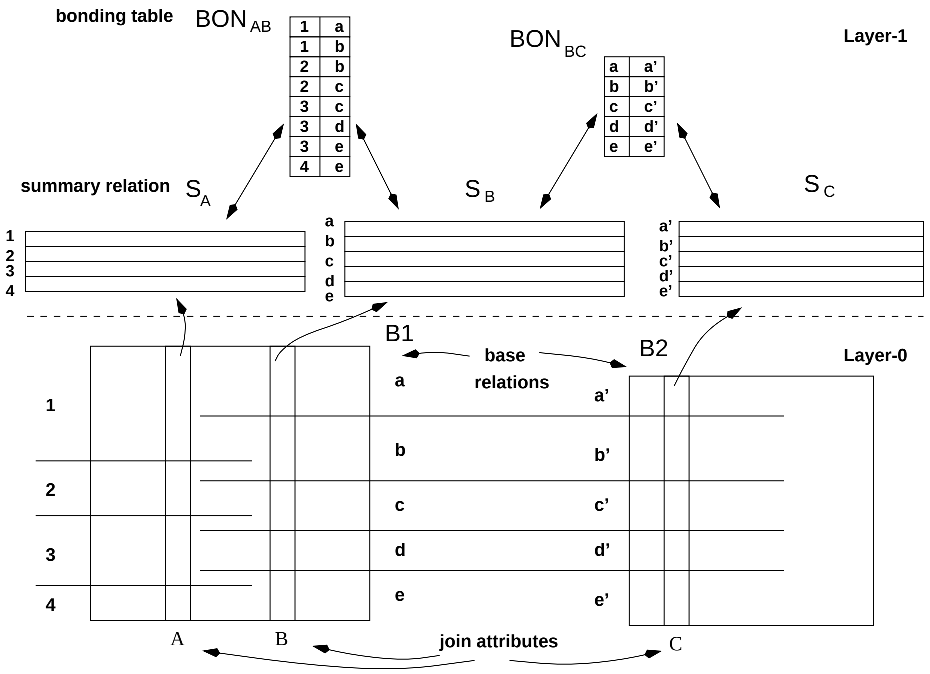

Figure 5: Bonding tables and summary relations.

In Fig.5, bonding table BON_{AB} links summary relations S_A and S_B which are from the same base relation $B1$, and bonding table BON_{BC} links summary relations S_B and S_C which are from different base relations $B1$ and $B2$, respectively. $B1$ and $B2$ are joinable through join attributes B and C. Horizontal line segments on the base relations $B1$ and $B2$ in Fig.5 represent groupings of tuples in base relations. Each summary tuple in S_A, S_B or S_C has a PID and is a tuple of group signatures. Each group signature is superimposed from a group of individual signatures as stated previously. As an example, tuple 2 in summary relation S_A corresponds to tuples b and c in summary relation in S_B, and this is recorded in the bonding table BON_{AB}. The PID of a summary tuple is not physically stored in the summary tuple.

Every tuple of a bonding table has two entries, each is a (logical or physical) address of a summary tuple in one of the two summary relations that are linked by this bonding table. If a tuple of a bonding table links two summary tuples in summary relations with different base relations, the two summary tuples have identical PID.

Let us take the following query example in conjunction with Fig.5 to illustrate a query execution in summary relations using bonding tables (relation $B0$ is not shown in Fig.5):

$$\begin{array}{ll} select & B1.name \\ from & B0,\ B1,\ B2 \\ where & B0.M = B1.A\ and \\ & B1.B = B2.C\ and \\ & B2.category = \text{``}autoparts\text{''}. \end{array}$$

If only summary tuples b' and c' in S_c satisfy predicate "autoparts", then through bonding table BON_{BC}, the join result in summary relations S_B and S_C can be determined as {bb', cc'}, in which the juxtaposition of symbols represents the concatenation of tuples. Note the semantic of a join in the summary relations: a joined tuple bb' indicates that some or all of the base tuples of b in $B1$ may be joinable with some or all of the base tuples of b' in $B2$. By executing a join between S_B and S_C, all tuples in $B1$ and $B2$ that belong to base groups of a, d, e, a', d', and e' can be excluded from the join execution. This achieves an effective pruning for the join execution in the main database.

Bonding table BON_{AB} links two summary relations that share the same base relation. After the execution of the join $B0.M = B1.A$, suppose that summary tuples 3 and 4 in the summary relation S_A were found to be relevant to the query. Summary tuples 3 and 4 and {bb',cc'} will be joined by using the bonding table BON_{AB} to obtain the query result {3cc'} from the summary relations. Then, {3cc'} is used to extract base tuples from $B1$ and $B2$ and be further joined. The execution of joins in the SDB greatly reduces the number of tuples in the main database that need to be considered for the join execution.

In short, bonding tables (such as BON_{AB}) between summary relations of the same base relation are used to connect execution results of several joins and/or selections of a query.

If a query only contains join predicates, then only bonding tables are used in the query execution and signatures in the summary relations are not used. Signatures are involved in the query execution only if a query contains selection predicates. For executing a complex query containing joins as well as selections, or containing multiple selections, both bonding tables and signatures in summary relations are used.

The partitioning of S_D affects the size of bonding tables and thus has a direct impact on the speed of join executions in the SDB. The join index is not widely used because its data structure may become excessively large [7]. One may have similar concern about the size of bonding tables. However, unlike the join index [5], bonding tables are built between summary relations which are much smaller than main database relations. If the average size of each group is 40 and the size of a signature is equal to the average size of an entry in the main database, then the size of a summary relation is only 2.5% of the size of its base relation. In addition, partitioning is an operation that can be controlled to some extent by database designers to reduce the size of bonding tables. Details of relevant strategies for reducing the size of bonding tables through proper partitioning will be given in Section 3.

Addressing tables: After a query execution is done in summary relations, a set of summary tuples are found to be relevant to the query. Some mechanism must be provided to map the result of the query execution from summary relations to the main database. We use a data structure called *addressing table* for this mapping. Conceptually, each entry in an addressing table has two entries: the logical address of a summary tuple and a set of logical addresses of the base tuples in the base group of its base relation.

We found that a B_c tree [7] is ideal to act as a part of an addressing table. The B_c tree method itself is better than the join index method. The B_c tree method uses much smaller data structure than the join index and the search of the B_c tree is much faster than the search of the join index partially due to the much smaller size of the B_c tree.

We can put (logical or physical) addresses of summary tuples in non-leaf nodes and (logical or physical) addresses of base tuples in leaf nodes of the B_c tree. A B_c tree is a modified B+ tree. Each leaf-node of a B_c tree contains a unique value from a data domain and several TID (tuple pointer) lists. Each TID is a (logical or physical) address of a base tuple that has this value as its join attribute value and all TIDs in a list point to such tuples in the same base relation. Out of each data domain (of join attributes) one B_c is is built. A B_c tree can be used to do fast join executions. Note that the number of B_c trees is equal to the number of data domains of join attributes in the main database.

With this implementation, leaf nodes of an addressing table can be used to complete the join execution started in summary relations. The query execution in summary relations only serves to reduce the search effort to find the query result in the main database and further processing is needed to determine the exact set of tuples of the main database in the query

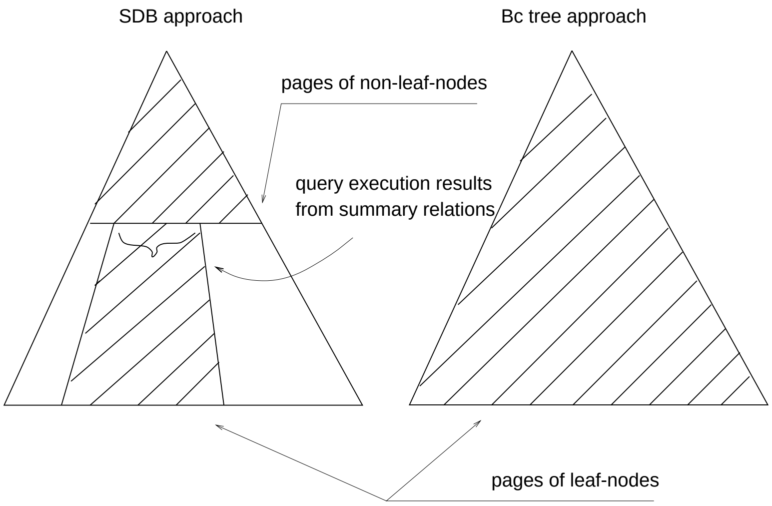

Figure 6: Search space reduction of an addressing table in a data domain.

result. Each attribute referred in a query corresponds to a data domain. The addresses of some summary tuples as the result of the query execution in summary relations are used to selectively load relevant disk pages containing leaf nodes of the B_c tree of the relevant data domain into the memory. Compared to the B_c tree method, the SDB method further reduces the number of disk I/Os needed in the join execution.

For example, if j_s summary tuples are the result of the query execution in summary relations and the average group size is 40, then the maximum number of relevant leaf nodes is $40 * j_s$. If we assume a disk page can contain several leaf nodes, then the maximum number of disk pages to be retrieved due to accessing of these leaf nodes is also roughly $40 * j_s$. This number is much less than the number of all disk pages containing leaf nodes of a B_c tree for a data domain of an join attribute in a very large database. This situation is sketched in Fig.6.

Summary relations, bonding tables, and addressing tables constitute a summary database (SDB). If a query contains only join predicates, than the query can be answered by a query execution completely within the SDB. If a query contains selection predicates, then after executing the query in the SDB, further execution is needed in the main database. For example, if a query contains *(J AND S)*, where J is a join predicate and S is a selection predicate, the join result of J is $R(J)$ and the selection result of S is $R(S)$, then the execution result T from the SDB satisfies $R(J) \cap R(S) \subseteq T \subseteq R(J)$. If $R(J) - T \neq \emptyset$, then $R(J) - T$ represents a pruning of the join result $R(J)$ due to executions in the SDB.

Generalization of the SDB: Group signatures in summary relations can be further grouped and superimposed to form even higher layer group signatures. A higher layer summary relation is much smaller than a lower layer relation, and thus, can be searched even faster. Because of the small size of the SDB, it is also possible to duplicate the SDB. Different copies of the SDB can share the same addressing table. Several copies of a SDB can be kept in the same layer to allow multiple queries to be executed parallelly: each query is executed sequentially while parallel executions are done among multiple query executions.

Heuristic bonding tables: The query execution result from a database equipped with a SDB is exact and complete. However, in some cases, such as online databases directly serving customer queries for product and service information, a strict, exhaustive search for every piece of information requested by the query may not be necessary and users may be satisfied if sufficient information is returned by a query. Often an individual user does not have enough time, nor the desire, to fully examine a large volume of information returned from an exhaustive-search query processing. In this type of application, fast query processing may be more desirable than the completeness of the query results.

This scenario makes further speed up of query executions in SDB possible. Since the size of bonding tables is important to the query execution performance in a SDB, we can further speed up query executions in a SDB by only keeping the most frequently used tuples in bonding tables.

In Fig.5, if a large number of queries reference link $(2, b)$ and almost no queries reference link $(1, b)$, then link $(1, b)$ can be removed from the bonding table BON_{AB}. The smaller incomplete bonding table that resulted from the removal of less frequently used links is called a *heuristic bonding table.*

A threshold value of the number of times a link is referenced can be used to filter out less frequently used links from a bonding table. Both heuristic as well as regular bonding tables can co-exist in a SDB. If a user of the database indicates that the completeness of the query result can be sacrificed for fast query processing, then heuristic bonding tables are used.

Cost of constructing the SDB: The cost of constructing the SDB is a few scans over the entire database. If the database is not frequently updated (frequent database updating is bad for any indexed method, not just the SDB method), the cost of constructing a SDB should not be a problem in practice.

We emphasize that the construction of summary relations and bonding tables is not done at the time of a query execution. Once the SDB has been built, any number of queries can be executed without incurring any additional cost of updating it as long as values in the main database are not updated. Therefore, the cost of constructing the SDB is not in the cost of query executions. From this discussion we can see that the SDB method is more suitable for relatively stable (with infrequent updates) and very large databases, for the cost of building a SDB can be amortized over a large number of queries.

We also emphasize that the query result obtained from the database with the assistance of the SDB is exact and complete. However, if strict, complete query result is not required, a "lazy" update policy for the SDB can be adopted. Small amount of updates to the main database do not have to cause immediate updates of the SDB. The SDB can be updated periodically. A relatively large time interval between updates can achieve a reduction in the cost of maintaining the SDB. In this sense, the SDB method is also suitable for query processing in data warehouses.

Dealing with attributes that have numerical data domains: The SDB method can be used to answer queries not only in attributes containing text values, but also to answer range queries in attributes containing numerical values. A numerical data domain D can be partitioned into many intervals. Every interval contains approximately the same number of numerical attribute values. An interval signature can be made out of each interval. Suppose the signature has length b. Two integers $0 < k_1 < k_2 < b$ are selected. The interval signature has exactly k_2 bits of 1 and each signature made from an individual numerical value has exactly k_1 bits of 1. All signatures made from individual numerical values must have their 1-bit be matched by 1-bit in the same position in the interval signature to ensure that the interval signature represents all numerical values in the interval.

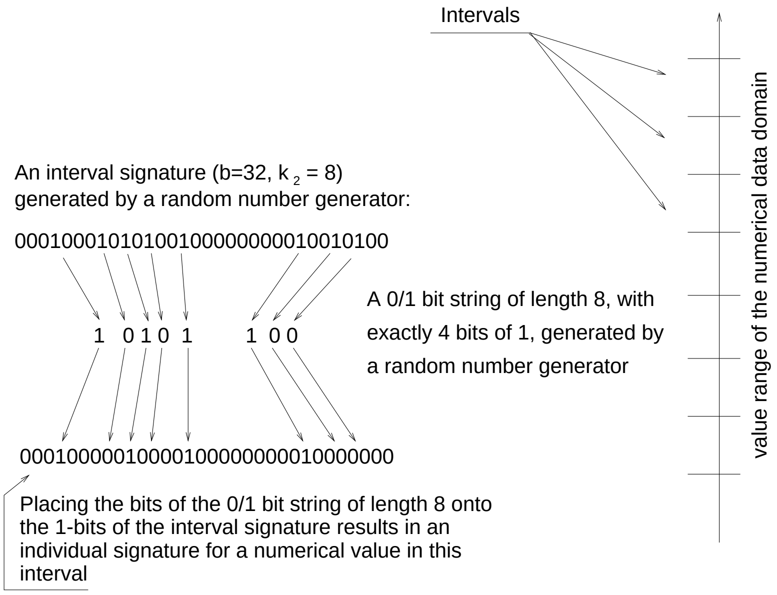

Figure 7: Construction of individual signatures within an interval.

Suppose there are m_1 numerical values be partitioned into m_2 intervals, $m_1 > m_2$. First a random number generator can be used to generate m_2 interval signatures of length b with exactly k_2 bits of 1. Then, for numerical values in each interval, a hash function can be used to hash each value in that interval into 0/1 bit strings of length k_2 with exactly k_1 bits of 1. The signature for each numerical value can be made by placing the bits of the generated

0/1 bit string onto the positions of the 1-bit in the interval signature as shown in Fig.7. The signature constructed in this way is always matched by the interval signature. In other words, if an individual signature can not match an interval signature, then the numerical value from which this individual signature is hashed can not lie in the interval represented by this interval signature.

If a join attribute of a base relation has at least one numerical value falls into an interval, then the interval signature of that interval is entered into the summary relation of that join attribute. For a query with predicates involving numerical attributes, values in the numerical predicates are made into query signatures using the procedure described above. The query execution is the same as that for the text valued attributes.

The partitioning granularity is a trade off between the pruning effect of the SDB and the complexity of constructing a query signature. For example, to partition the whole range of a numerical data domain into 1,000 intervals can achieve a better pruning effect than to partition it into 100 intervals. But to partition the whole range of a numerical data domain into 1,000 intervals means that the query processor has to keep an array of 1,000 interval signatures in order to translate the numerical values in query predicates into query signatures.

3 Cascaded Partitioning Algorithm

Different grouping of tuples in the main database will result in different degree of interconnecting complexity between summary relations and this in turn can result in different size of bonding tables. A good partitioning algorithm to group tuples in the main database can result in a small overall size of all bonding tables in the SDB. In Section 2, we described a way of reducing the size of bonding tables between summary relations of different base relations. In this section, we will describe how to reduce the size of bonding tables between summary relations of the same base relation and a partitioning algorithm for constructing small bonding tables.

Let R be a relation with two joinable attributes $A1$ and $A2$. The partitioning of values in $A1$ results in a summary relation $S1$ and the partitioning of values in $A2$ results in a summary relation $S2$. We may choose to partition values of $A1$ first. The partitions of values in $A1$ naturally induces a partitioning of tuples of R, and hence the partitioning of values in $A2$. This results in a one-to-many relationship between tuples of $S1$ and tuples of $S2$. The resulting bonding table linking $S1$ and $S2$ has no more tuples than the number of tuples in R.

This partitioning algorithm can reduce the size of bonding tables. In this section, we go into details of this partitioning algorithm and answer the question "which one, $R1$ or $R2$, should be chosen as the relation to be partitioned first?" and "which one, $A1$ or $A2$, should be chosen as the join attribute to be partitioned first ?" in a systematic way. The name *cascaded partitioning* comes from the fact that relations are partitioned, one by one, in a sequential order, and if relation $R2$ is partitioned after relation $R1$, the partitioning of $R2$ cannot start until the partitioning of $R1$ has been completed.

Ranking join attributes: Obviously, larger and more important relations should be partitioned before smaller and less important ones. Common sense like this leads to a ranking of all join attributes in the main database based on the following four principles:

- P1: more important attributes are ranked higher. For example, in a supermarket data warehouse, 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 size of clothes is ranked higher than the description of clothes, and the address of a customer (which can be geographically grouped) is ranked higher than the name of the customer.
- 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.

These principles are used to sort join attributes in an order to resolve conflicts in the priority of grouping of tuples in main database relations. Conflicts arise because a main database relation may have more than one join attribute and groupings of tuples based on groupings of values in different join attributes may, in general, be different from one another. These conflicts cause an undesirable many-to-many relationship between tuples of different summary relations that share the same base relation, resulting in a large bonding table linking these summary relations.

Join attributes in larger relations (P4), more important join attributes (P1), more useful join attributes (P2), or more easily partitioned join attributes (P3) are assigned higher ranks to allow them to be partitioned first. These principles are heuristics and may be in conflict with each other when applied in practice. Should that happen, common sense and business knowledge should be applied in resolving those conflicts.

These principles only help to resolve the problem when two join attributes are compared and a decision has to be made to choose which of them is the join attribute to be partitioned first. To clearly illustrate the order of partitioning among a large number of join attributes, we need the assistance of a concept which we call *linear ordering*.

Linear ordering: A linear ordering LO is defined as a connected, directed non-cyclic graph. Each node of LO represents a join attribute. If two join attributes A_1 and A_2 are in the same main database relation, or they share the same data domain, then there should be a directed edge E linking A_1 and A_2 in a LO. A database may have many LOs.

If A_1 is ranked higher than A_2, then E is from A_1 to A_2 and we say that A_2 follows A_1. A linear ordering LO consisting of join attributes A_1, A_2, ... , A_k with decreasing ranks can be denoted as $LO = (A_1, A_2, ..., A_k)$.

Both the way to derive linear orderings and the derived linear orderings may not be unique. This is because the ranking principles are heuristics and when there are conflicts in applying these principles, the way to resolve the conflicts are not unique. However, the important thing is that every join attribute should be included in a linear ordering.

Cascaded partitioning algorithm: Partitioning of join attribute values is done starting from the highest ranking join attribute A_1 of a relation R in a linear ordering $LO = (A_1, A_2, ..., A_k)$. LO is picked arbitrarily among several linear orderings. Let D_1 be the data domain of A_1 and S_{D1} be the set of all values of all join attributes that share D_1. Note that $S_{D1} \subseteq D_1$. Values in S_{D1} are partitioned. If A_1 is a join attribute of relation R, the partitioning of S_{D1} will induce a partition of R. The partitioning process should try to ensure that the induced partitioning (of tuples in R) is even.

The "evenness" should be applied towards the partitioning of tuples, not the partitioning of the distinctive values in the join attribute. For example, in the partitions of S_{D1}, one partition may have 38 distinct join attribute values and another partition may have 17 distinct join attribute values. Each partition of join attribute values induces a partition of tuples in R. The two induced partitions of tuples in R should have approximately the same number of tuples.

We would like to mention a strategy to evenly partition a set of values [16]. In order to evenly partitioning S_{D1} into k partitions R_1, R_2, ... , R_k, first n (n is much larger than k) random samples are taken from S_{D1} and are sorted. Let s_1, s_2, ... , s_n be the list of sorted values in ascending order. Then $r_{n/k}$ is the partitioning value between R_1 and R_2, and $r_{2n/k}$ is the partitioning value between R_2 and R_3, etc. Of course, this approach can be used only if S_{D1} can be linearly ordered and the even partitioning of values can ensure an even partitioning of tuples.

Now, let us come back to the discussion of cascaded partitioning algorithm. Let S_{A_1} denote the set of all values in A_1 and $PD_1 = \{p_1, p_2, ..., p_k\}$ be a partitioning of S_{D1} ($S_{A_1} \subseteq S_{D1}$). S_{A_1} can be partitioned as $PS_{A_1} = \{S_{A_1} \cap p_1, S_{A_1} \cap p_2, ..., S_{A_1} \cap p_k\}$. PS_{A_1} can induce a partitioning of tuples in R. This partitioning (along with summary information extracting) results in the construction of one summary relation.

If A_2 is a join attribute following A_1 in a linear ordering LO, the partitioning of S_{D2} (all values in the main database in data domain of A_2) can be classified into two cases (let the partitioning of S_{A_1} be denoted as $PS_{A_1} = \{pp_1, pp_2, ..., pp_k\}$):

1. If both A_2 and A_1 are in the same main database relation, then there is a induced partitioning of S_{A_2} (the set of all values in A_2) by PS_{A_1} through the induced partitioning of tuples of R.

   Formerly, the induced partitioning of S_{A_2} is $PS_{A_2} = \{pp'_1, pp'_2, ..., pp'_k\}$ and $pp'_i = \{t[A_2] \mid t[A_1] \in pp_i, tisatuple\}$, $1 \leq i \leq k$. Then, PS_{A_2} can be further refined to complete the partitioning for S_{A_2}, as discussed in the beginning of this section.

2. If A_2 and A_1 are in two different main database relations, then by the definition of the linear ordering, $D_2 = D_1$. The partitioning of S_{D2} has already been done. The partitioning of S_{A_2} is $PS_{A_2} = \{pp'_1, pp'_2, ..., pp'_k\}$ and $pp'_i = \{v \mid v \in (S_{A_2} \cap pp_i) \cup d_i\}$, $1 \leq i \leq k$. $\{d_1, d_2, ..., d_k\}$ is a partitioning of some values in S_{A_2} that are not in S_{A_1}. Note that $S_{A_2} = \cup_{i=1,...,k} (pp_i \cup d_i)$, and $(pp_i \cup d_i) \cap (pp_j \cup d_j) = \emptyset$, if $i \neq j$.

This process is repeated for each node of LO until the entire linear ordering is processed. Each linear ordering is processed in this way. Eventually, all join attributes are processed 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 two summary relations that are linked by this bonding table. Every tuple in a bonding table has only two entries containing addresses of summary tuples, and an address is usually smaller in size than a signature. So, the size of a bonding table is usually much smaller than the summary relations it links.

The partitioning of values in join attributes can also be helped by using concept hierarchies. Concept hierarchy [10, 12, 15, 17] is a tree-like structure. Each concept is represented as a node in the concept hierarchy (tree) and the relationship between a node and its parent node denotes an "is-a" relationship. Concept hierarchies are application specific and can be provided by domain experts to database designers for constructing a SDB.

In actual implementation, a special attention should be paid to object IDs such as employee IDs, product IDs, etc. These types of "ID" attributes are frequently used in commercial implementation of relational databases as the key attributes of relations, and often they are join attributes. "ID" itself does not convey any meaningful concept so a proper partitioning of the value set S_A of an "ID" attribute A seems difficult. One way of dealing with this difficulty is to select another attribute T (whose values can be partitioned more meaningfully) in the same relation R as A, and to partition S_T. The partitioning of S_T can induce a partitioning of tuples in R, and that in turn can induce a partitioning of S_A.

4 Discussion and Related Work

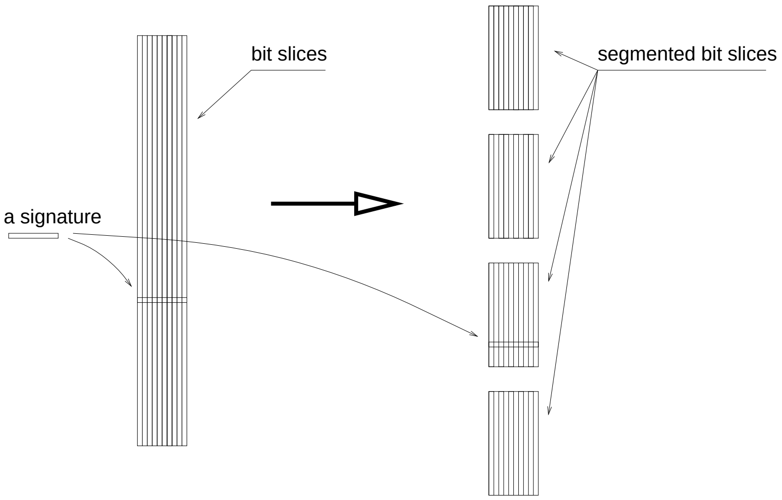

Figure 8: A bit-sliced storage format and its segmentation.

Using bit-sliced storage format: A bit-sliced format for storing signatures [3, 4] can be used for summary relations. Instead of storing signatures, the bit-sliced format stores all bits in a particular position of all signatures as a slice, as shown in Fig.8. If the length of signatures is 200 bits and there are 2,000 signatures, then 200 bit slices of length 2,000 are stored. When a query signature is presented to the summary relation, only those bit slices in the summary relation that correspond to the positions of 1-bits in the query signature need to be retrieved. For example, if only 4 1-bits in a query signature and rest are all 0-bits, then only 4 bit slices are retrieved. This represents a reduction in scanning of the summary relation by (200 - 4)/200 = 98%.

A bitwise AND operation is performed for these retrieved bit slices and a 0/1 bit string of 2,000 bits is obtained. Each 1-bit in this resulting 0/1 bit string indicates a signature that matches this query signature. For example, in the resulting 0/1 bit string, if only the 124-th and 875-th bits are 1 and rest are all zero, then the 124-th and 875-th signatures are found to match the query signature.

The bit-sliced format reduces search effort for query processing but has high updating cost since slices are usually very long [13]. A segmented approach can be used to reduce the cost of maintaining bit-sliced signature files as shown in Fig.8. Each bit slice is split into multiple slice segments so that only one segment is involved when inserting or deleting a signature. The search effort in a signature file remains the same as that of the non-segmented version. In the above example, if a bit slice of 2,000 bits is split into 20 segments, then each segment only has 100 bits and the updating cost reduces to 5% of the non-segmented version. Other fast searching signature methods such as the two level signature file [4] may also be used.

The two level signature method is found [21] to be faster than other fast searching signature files such as indexed descriptor file and (fixed or floating prefix) segmented signature files. All these fast searching signature file methods have higher maintenance cost and we will not elaborate on them in this paper. [21] provides a survey and performance evaluation for these signature files.

Comparison to using tuple signatures: One may raise a question: why not use tuple signatures to facilitate query executions? A tuple signature [6] is a signature superimposed from individual signatures of all entries of a tuple. In practice, there are several problems associated with the use of tuple signatures. Since a tuple signature is superimposed from individual signatures of all entries of a tuple, the length of the tuple, i.e., the number of attributes in a schema, affects the quality of the tuple signature. In general, a relational database have many different relations, each having a different number of attributes. A relation with a larger number of attributes needs to use longer signatures to avoid saturation of signatures, i.e., situation when most of the bits in a tuple signature are set to 1. A relation with a smaller number of attributes can use shorter signatures. All relations may adopt a common length of signatures suitable for the relation with the largest number of attributes. But this is obviously an inefficient approach. Also, entries in a tuple are from different data domains and their signatures are obtained using different hash functions. In practice, how to select and adjust hash functions so that the 1-bits of one entry is not dominant over 1-bits of other entries in the tuple signature is a non-trivial problem.

Most importantly, the SDB method groups signatures along each attribute, and this is the only way that superimposed signatures can be used in join execution. Simply explained, because signatures are grouped and superimposed along each attribute, the signatures of join attributes are separated from signatures of other attributes and therefore they can be used to represent join attribute values to execute joins.

Comparison to join index and B_c trees: Indexes (e.g., B+ tree) that were originally designed for selections, can be used to speed up join executions [9, 11]. However, the SDB method belongs to those cross-relational indexed methods that are specially designed for the execution of joins such as the access path index [2], join index [5], and B_c tree [7].

A performance comparison among the hybrid hash join, materialized view, and join index [8] found that when the data size is large and the selectivity is low (small join result), the performance of the hybrid hash join is the worst, the materialized view is better, and the join index is the best. So, we only consider the join index and B_c tree method here.

Among cross-relational indexed methods, B_c tree method is the closest to the SDB method because the B_c tree is also a cross-relational index employing B+ trees. The B_c tree method improves over the join index method in two aspects: the data structure is much smaller and B+ trees are used. In the join index method, the number of join indexes may be as high as C_2^m (combination number of choosing 2 out of m), where m is the number of relations in the database. In the B_c tree method, the number of B_c trees is equal to the number of data domains in join attributes which is usually less than the number of join attributes. In the join index method, the number of tuples in a join index may be as high as the cross-product of number of tuples in both joining relations. In the B_c tree method, the number of entries of all leaf nodes of a B_c tree is about $n_1 + n_2$, where n_1 and n_2 are number of tuples in each of two joining relations, respectively. Since all leaf nodes constitute the bulk of a B_c tree, the size of a B_c tree is about $(n_1 + n_2) * TID_{sz}$, where TID_{sz} is the size of a TID (tuple ID).

To appreciate the size difference between the data structures used by the B_c tree method and that used by the join index method, we can see that the number of indexing structures in the join index method is $O(m^2)$, while that in the B_c tree is $O(m)$; the size of each indexing structure in the join index method is $O(n_1 * n_2)$ while that in the B_c tree method is $O(n_1 + n_2)$. This analysis explains why we adopted the B_c trees as part of our addressing tables.

For a full join (nonselective join, i.e., all join attribute values are possible candidates of matching values [7]) execution alone, our SDB method improves over the B_c tree method in that not even all the pages of leaf nodes of the B_c are needed in the join execution.

The B_c method is a join method. Our SDB method is not just a join method. One can execute a query containing multiple joins and/or selections in a SDB. To be more specific, queries can be executed faster in databases equipped with a SDB than databases equipped with B_c trees in two cases: (1) joining relations are large and join selectivity is low; (2) query has multiple joins and/or selections.

When the joining relations are large, for a full (nonselective) join, even a full scan of all the leaf nodes of a B_c tree, as in the B_c tree method, is costly. The join result from the SDB can be used to selectively load disk pages containing relevant leaf nodes of the B_c tree into the memory for the join execution and the cost of disk I/Os is reduced. The reduction is very effective when joining relations are very large.

Assume each of the two joining relation has 10^5 tuples; the true join result has 100 tuples, joined from 100 tuples from each of two joining relations; the average group size in the construction of the SDB is 20. Then, the number of tuples in each of the joining summary relations is $5,000$. In the result of a join execution from the summary relations, the number of summary tuples from each joining summary relation will be no more than 100, provided that the false drop ratio in scanning signatures is very low. This is because each pair of matching summary tuples must contain at least one pair of matching base tuples as the worst case scenario.

The above discussion holds only when the false drop ratio is very low, such as 1 out of

10,000. Such a low false drop ratio is possible in practice. We can see that a false drop ratio of 1/10,000 can be achieved by using signatures of length 200 bit (see the analysis in Section 5).

100 summary tuples can be mapped to at most $100 \times 20 = 2,000$ base tuples in each of the joining base relations. So, only $2,000$ TIDs of each relation need to be checked for matches. This represents a reduction in disk I/Os from the B_c tree method by: $(10^5 - 2,000)/10^5 = 98\%$.

If a query contains multiple conjunctive join and/or selection predicates, the execution of each predicate in the SDB will cause a search space reduction in the main database, and the combined pruning effect will be greater than a single predicate. So, the SDB method is better suited for answering complex queries having multiple join and/or selection predicates, such as typical OLAP type of queries in data warehouses.

A SDB as a selection index: The SDB method is more than just a join method. A query having multiple join and/or selection predicates can be executed in a SDB. The execution result of a query containing selection predicates can be used to restrict the search space of selection in the main database and thus, speeding up selection executions. It is well known that a fully inverted file requires very large storage space for its indexing structure, sometimes even larger than the data file itself. In lieu of the large storage cost of setting up one B+ tree index for each attribute in the database, the SDB is a light-weighted index for indexing all attributes in selection executions.

Due to the space limitation, a precise performance comparison of the SDB method with other commonly used selection indexs for selection only queries is beyond the scope of the paper and we intend to present the result elsewhere. As a rough estimate, we know that the false drop ratio of 1/10,000 can be achieved by using signatures of length 200 bits (see the analysis in Section 5). This excellent filtering effect of signatures, plus the reduction of signature scanning by using bit-sliced format (see discussion at the beginning of this section) or by simply storing all summary relations and bonding tables in the memory (this is possible since together, they take less than 20% of the space needed by the main database; see discussion in Section 5), allows the SDB method to have a comparable performance to a fully-inverted-file-per-relation approach, which is a very storage demanding approach.

5 The Size of a SDB and its Pruning Effect

The size of all data structures in the SDB method is larger than the size of all data structures in the B_c tree method. All addressing tables take about the same storage space as all B_c trees in the B_c tree method. The size of all addressing tables is of the order of $O(m * n * TID_{sz})$, where m is the number of base relations, n is the average number of tuples in a base relation,

k	number of 1-bits in an individual signature, assume $k = 4$
b	length (number of bits) of a signature
s	group size in construction the SDB, assume $s = 40$
q_j	number of conjunctive join predicates in a query
q_s	number of conjunctive selection predicates in a query
e	average number of bytes in a database entry, assume e=6
F	false drop ratio of using signatures

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

and TID_{sz} is the size of tuple IDs (see discussion in Section 4).

In addition, the SDB method uses summary relations and bonding tables. The length of signatures determines both the pruning effect of using a SDB as well as the size of summary relations. Long signatures have low false drop ratio while they need large storage space. Using the symbols listed in Table 1, the formula for computing false drop ratio of searching a signature file using a query signature hashed from a single predicate is [4]:

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

A query may have multiple selection and join predicates. Taking into account the fact that a join involves two summary relations, we obtain the false drop ratio for a query having q_j conjunctive join predicates and q_s conjunctive selection predicates as:

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

The number of selection and join predicates in queries are varied in equation (2) to compute false drop ratios shown in Fig.9. From these results, it is clear that the use of the SDB method is more effective in answering complex queries having multiple selection and/or join predicates, especially multiple join predicates.

We also conducted simulations to verify these analytical results. In simulations, we kept n = 10,000, which is the total number of different group signatures. So, the total number of values is 400,000. Simulation results were averaged over repeated runs and they came very close to the analytical results (see Fig.9). In most cases, the difference is only 1 out of 10,000, or even lower. The analytical results slightly underestimated the number of false drops than the simulation results. The cause of this slight difference is that, in equation (1), the term $(1-k/b)$, which is an approximation of the probability that a 1-bit does not appear at a particular position in a signature, is slightly larger than its actual value if a signature has multiple 1-bits.

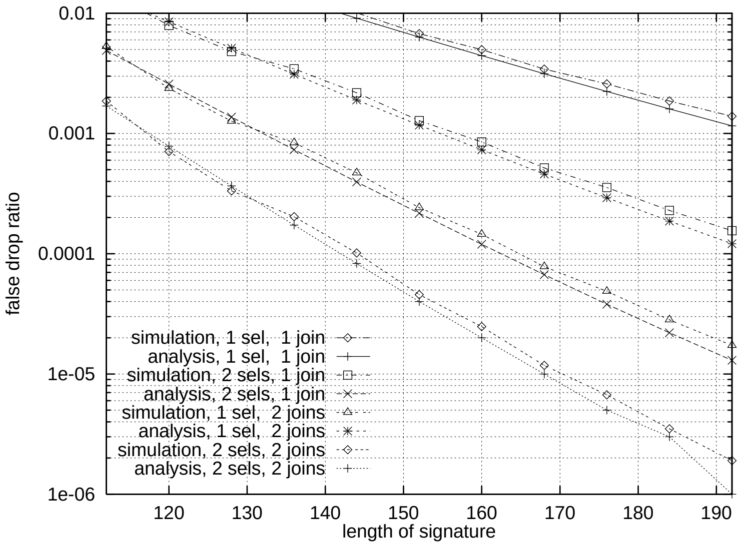

Figure 9: Signature lengths and false drop ratios.

The length of signatures directly determines the size of summary relations. Fig.10 shows the relationship between the computed false drop ratios of searching signatures and the overall size of all summary relations.

The storage space requirement of a SDB is the size of all addressing tables, plus the size of all summary relations, and plus the size of all bonding tables. The size of all addressing tables is about the same as the size of all B_c trees in the B_c tree method. From the analytical and simulation results, the size of all summary relations is about 5% to 10% of the size of all base relations in the main database. The size of all bonding tables is much smaller (see discussions in Section 3) than the size of all summary relations.

For queries having selection predicates, the SDB method is an access index that takes much less storage space than the inverted file approach. For queries with join predicates, the SDB method is a much faster access index than the B_c tree method and certainly the join index, and the SDB method takes storage space larger than that of the B_c tree method but smaller than that of the join index.

6 Performance Comparison

In this section, we compare the join execution performance between the SDB method and the B_c tree method. From the discussions in Section 4, it is clear that the B_c tree method is the closest join method to the SDB method and the only join method which has comparable performance to the SDB method for low selectivity joins in very large databases. However, the SDB method is more than just a join method and in that sense the B_c tree method is not fully comparable to it. The performance of the SDB method is also compared with

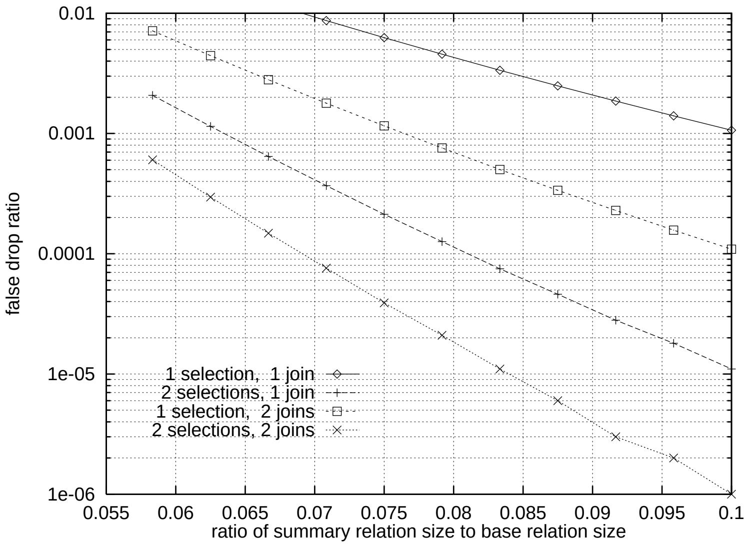

Figure 10: Summary relation sizes and false drop ratios.

hybrid hash join, which is one of the best and commonly used non-indexed methods. The comparison of these join methods is done under a common parameter setting.

As discussed in Section 4, the performance of the SDB method is especially good in answering complex queries having multiple join and/or selection predicates. However, due to space limitation and for ease of explanation, we use the following two simple queries for performance comparison:

$Q1:$

$$\begin{array}{ll} select & B1.name \\ from & B1,\ B2 \\ where & B1.B = B2.C; \end{array}$$

and

$Q2:$

$$\begin{array}{ll} select & B1.name \\ from & B1,\ B2 \\ where & B1.B = B2.C\ and \\ & B2.D = \text{“}electronics\text{”}. \end{array}$$

Query $Q1$ is used to evaluate the performance of executing a full join by various join methods. Query $Q2$ is used to evaluate the performance of executing a selective join by various join method.

$P_{sz} = 2,048$	page size (in bytes) of disk pages
$M = 5,000$	size (in # pages) of available memory space
$\mid B1 \mid$=200k	# pages of $B1$
$\mid\mid B1 \mid\mid$	# tuples of $B1$ ($= 5* \mid B1 \mid$)
α	α% tuples of $B1$ are joinable to tuples in $B2$
$\beta = 3$	each joinable tuple of $B1$ can join β tuples in $B2$
$g = 40$	grouping size (for constructing the SDB)
$b1_0$, $b2_0$	size (in pages) of anchor buckets of $B1$ and $B2$
$seq = 50\ pages$	$seq =\mid buf_{bucket} \mid$ in the hybrid hash join
j_s	join selectivity (in # tuples) in the SDB
$s_s = 20\%$	selectivity (in %) in the SDB due to the selection in SDB
$s_m = 80\%$	selectivity (in %) in main database due to selection in SDB
$s = 5\% \mid B2 \mid$	selection selectivity (in # pages) in the main database
θ	disk access time ratio: $\theta = t_{random}/t_{sequential}$
$TID_{sz} = 4$	the size (in bytes) of a TID
$A = 8$	the average size (in bytes) of data entries

Table 2: Parameters used in the performance comparison.

Assumptions: Performance is measured by the number of equivalent sequential disk I/Os (the time cost of a sequential disk access is assumed to be 1 and of a random disk access is $\theta > 1$). Relations $B1$ and $B2$ are assumed to have the same average tuple size (= number of attributes $\times$ average size of data entries). On the average, each disk page can contain 5 tuples. All other parameters are listed in Table 2. The cost of the writing out of join results is not included in the comparison since it is the same for all methods.

Formulas for estimating the number of disk I/Os: The number of disk pages accessed when accessing k tuples randomly distributed in a relation with m pages and n tuples via a B+ index of two or three levels is given by [8]:

$$Y_{io}(k,m,n) = Yao(k,m,n) + Yao(k,n/FO,n) + T3(k,n), \tag{3}$$

where

$$T3(k,n) = Yao(Yao(k,n/FO,n), n/(FO*FO), n/FO). \tag{4}$$

The formula $Yao(*)$ gives the number of disk accesses without any index searching overhead, and is by Yao, S. B. [1]:

$$Yao(k,m,n) = m - m * \prod_{i=1}^{k} \frac{n-(n/m)-i+1}{n-i+1}. \tag{5}$$

Query execution cost in the hybrid hash join: Though there are more accurate cost formulas (such as [19], it uses a queuing model) for the hybrid hash join, because we only need a rough estimate of the join cost of the hybrid hash join and because of the large performance gap between the hybrid hash join and the other two indexed methods, we use a simple cost formula as described below. Some disk accesses of the hybrid hash join are nearly sequential, such as reading of pages of $B1$ and $B2$ into the memory for hashing and reading of hashed buckets of $B1$ and $B2$ into the memory for joining. The writing of hashed results out to hash buckets on the disk is more random and the extent of randomness depends on the size of the memory buffer for each hash bucket if we adopt the technique proposed in [22]. For example, if $\theta = 100$ and the memory buffer of each hash bucket is 50 pages ($seq = 50$, that is 100k bytes), then the average time to write a disk page when flushing out a buffer of hashed result to a bucket is $\theta_{bucket} = (\theta + 50 - 1)/50 = 2.98$. If there are total of 20 hash buckets, then $100k \times 20 = 2MB$ memory space is used for these buffers. This is not a significant portion of available memory space on modern computers. The disk I/O cost of the hybrid hash join for query Q1 is:

$$\begin{aligned} C_{hybrid\ \ hash(Q1)} = \quad & \mid B1 \mid + (\theta_{buckets} + 1)(\mid B1 \mid - b1_0) \\ & + \mid B2 \mid + (\theta_{buckets} + 1)(\mid B2 \mid - b2_0) \end{aligned} \tag{6}$$

The first two terms represent the time to read pages of $B1$ into the memory for hashing, flushing out (note the random access factor $\theta_{buckets}$) the hashed results to hash buckets to the disk (except the anchor bucket of $b1_0$ pages), and read them back into the memory for joining. Similar explanation can be given to the third and the fourth terms, with $B1$ be replaced by $B2$ and $b1_0$ be replaced by $b2_0$. $b1_0$ is set at $0.7 * M$ and $b2_0 = B2 * b1_0/B1$.

For query Q2, the selection costs additional $(\mid B2 \mid + s)$ (including reading of $B2$, and writing of the result back to the disk) disk I/Os but the cost of join execution reduces to:

$$\begin{aligned} C_{hybrid\ \ hash(Q2)} = \quad & \mid B1 \mid + (\theta_{buckets} + 1)(\mid B1 \mid - b1_0) \\ & + s + max\{0, (\theta_{buckets} + 1)(s - b2_0)\} \end{aligned} \tag{7}$$

Query execution cost in B_c: For query Q1, a full scan over all leaf nodes of a B_c tree is needed [7]. The query execution cost for Q1 is:

$$\begin{aligned} C_{B_c(Q1)} = \quad & (\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz} \\ & + \theta * Y_{io}(\alpha \| B1 \|, \mid B1 \mid, \| B1 \|) \\ & + \theta * Y_{io}(\alpha\beta \| B1 \|, \mid B2 \mid, \| B2 \|) \end{aligned} \tag{8}$$

The first term is the time to read in pages containing all leaf nodes (TIDs and join attribute values) of the B_c tree and the second and the third terms are the time to retrieve matching tuples from $B1$ and $B2$.

For query Q2, the selection costs additional ($\mid B2 \mid +s$) disk I/Os and in the join execution, only some selective leaf nodes (as a result of the selection) are accessed [7]. The cost of the join execution for Q2 is:

$$\begin{aligned} C_{B_c(Q2)} = \;& \theta * Y_{io}((5 * s + \parallel B1 \parallel), \\ & (\parallel B1 \parallel + \parallel B2 \parallel) * (A + TID_{sz})/P_{sz}, \parallel B1 \parallel + \parallel B2 \parallel) \\ & + \theta * Y_{io}(\alpha \parallel B1 \parallel, \mid B1 \mid, \parallel B1 \parallel) \\ & + \theta * Y_{io}(\alpha\beta \parallel B1 \parallel, s, 5 * s) \end{aligned} \tag{9}$$

The first term is the time to read selection results. The second term is the time to read the B_c tree. Note that the number of leaf nodes accessed is $5 * s$ or $\parallel B1 \parallel$, whichever is lower. The second and third terms are the time to read matching tuples.

Query execution cost of using the SDB: As discussed in Section 5, false drop ratio can be made very low by using relatively long signatures, such as 200 bits. So, equation (10) does not take into account of the false drops. Query Q1 has only a join predicate So, only a bonding table is searched. Bonding tables are much smaller than summary relations and are assumed to stay in the memory. For query Q1, the disk I/O cost in a database equipped with a SDB is:

$$\begin{aligned} C_{SDB(Q1)} = \;& min\{(\parallel B1 \parallel + \parallel B2 \parallel) * (A + TID_{sz})/P_{sz}, \\ & \theta * Y_{io}(min\{j_s * g, \alpha\beta \parallel B1 \parallel\}, \\ & (\parallel B1 \parallel + \parallel B2 \parallel) * (A + TID_{sz})/P_{sz}, \parallel B1 \parallel + \parallel B2 \parallel)\} \\ & + \theta * Y_{io}(\alpha \parallel B1 \parallel, \mid B1 \mid, \parallel B1 \parallel) \\ & + \theta * Y_{io}(\alpha\beta \parallel B1 \parallel, \mid B2 \mid, \parallel B2 \parallel) \end{aligned} \tag{10}$$

The first term denotes the time needed, after searching a bonding table, to selectively load pages containing leaf nodes of an addressing table (it is actually a B_c tree itself) into the memory to execute the join. This time depends on the join selectivity j_s in summary relations. From discussion in Section 4, j_s satisfies $\alpha\beta \parallel B1 \parallel /g \leq j_s \leq \alpha\beta \parallel B1 \parallel$. When the join selectivity in the main database is low, only part of $(\parallel B1 \parallel + \parallel B2 \parallel) * (A + TID_{sz})/P_{sz}$ pages are loaded into the memory. The second and third terms denote the time to read matching tuples.

Assume bit sliced format is used. If a signature has length 200 and four 1-bits are set in the query signature for each predicate value in the query, then for searching a signature file, only four out of 200 bit slices are retrieved. This represents a signature search reduction of (200-4)/200 = 98%. Also, a signature has 200 bits and a data value (see Table 2) has 8 bytes = 64 bits, thus the size of a signature is 200/64 = 3.125 times that of a data value. So, for query Q2, $3.125* \mid B2 \mid /(50 * g)$ pages of bit slices are retrieved.

For query Q2, the number of matching tuples to be retrieved will be less than that of Q1 due to the pruning effect of selection execution in the SDB. The selection does not add any additional execution cost since the selection can be done in retrieved matching tuples. The cost of the query execution for Q2 is:

$$
\begin{aligned}
C_{SDB(Q2)} = \quad & (3.125* \mid B2 \mid)/(50 * g) \\
& +min\{(\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz}, \;\; \delta\} \\
& +\theta * Y_{io}(j_s * s_s * *s_m * g, \mid B1 \mid, \| B1 \|) \\
& +\theta * Y_{io}(j_s * s_s * *s_m * g, \mid B2 \mid, \| B2 \|)
\end{aligned}
\tag{11}
$$

where

$$
\begin{aligned}
\delta = \quad & \theta * Y_{io}(min\{j_s * s_s * s_m * g, \alpha\beta \| B1 \|\}, \\
& (\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz}, \| B1 \| + \| B2 \|)\}
\end{aligned}
\tag{12}
$$

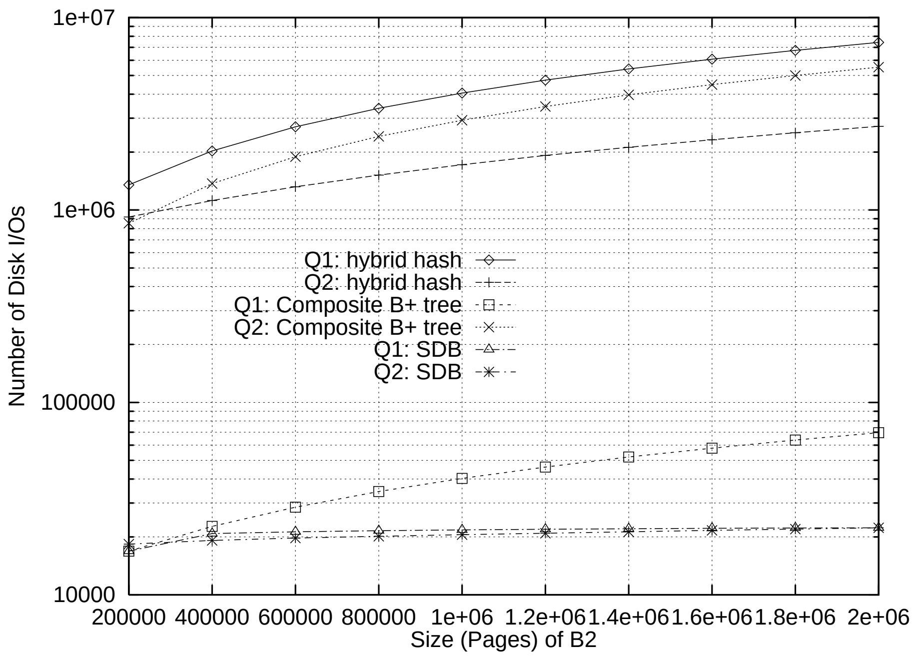

Figure 11: $\mid B1 \mid= 200k$ pages; $s = 10k$ pages; $\alpha = 0.0001$; $\theta = 20$.

Numerical results: The performance (measured by the number of disk I/Os required) of the hybrid hash join, the B_c tree method, and the SDB method is plotted in Fig.11, Fig.12, and Fig.13. The value of input parameters is shown in Table 2. The worst case performance of the SDB method is plotted in these figures. For example, if $\mid B1 \mid= 200k$ pages (i.e., 1000K tuples) and $\alpha = 0.0001$, then the true join result should contain 100 tuples from $B1$, and the number of summary tuples (in $B1$'s summary relation) resulted from the query execution in summary relations (without considering false drop) should be no more than 100. So, $j_s = 100$ represents the worst case scenario (out of every group of 40 base tuples

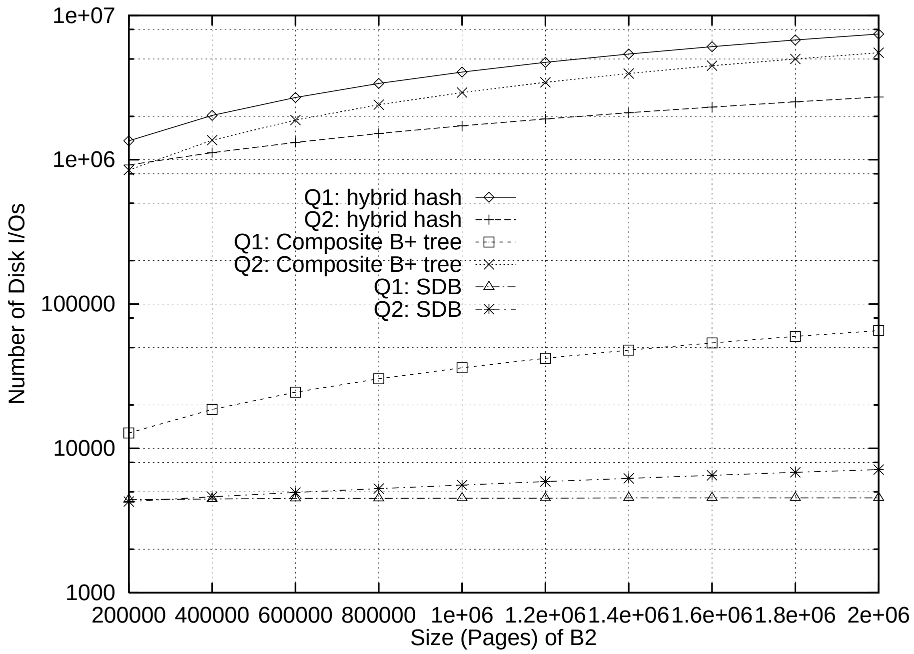

Figure 12: $\mid B1 \mid = 200k$ pages; $s = 10k$ pages; $\alpha = 0.00002$; $\theta = 20$.

of a matching summary tuple, only one base tuple is joinable). In practice, the worst case rarely occurs and there will be (even the probability can be made very low) false drops.

In Fig.11 and Fig.12, the size of $B2$ varies from $\mid B1 \mid$ to $10 \mid B1 \mid$. The difference between Fig.11 and Fig.12 is in different join selectivities: $\alpha = 0.0001$ in Fig.11 and $\alpha = 0.00002$ in Fig.12. Note that relative selectivities $s_s = 20\%$ and $s_m = 80\%$ are conservative estimates $(20\% * 80\% = 16\%)$ because $s = 10k$ means that the relative selectivity in $B2$ is less than or equal to 5%.

From Fig.11 and Fig.12, we observe that with or without a selection, the SDB method consistently performs better than the B_c tree method and the hybrid hash join. Both indexed methods, B_c tree and SDB, perform better in Fig.12 than in Fig.11 because of lower join selectivity. Our result of the performance of the B_c tree method is somewhat worse than the result in [7] because we take into account the time difference between a random disk access and a sequential disk access. The B_c tree method performs much worse for query Q2 than for query Q1 because the B_c tree method is only a join method.

Fig.13 shows the influence of θ (random access factor) on the performance of three join methods and uses the same setting of input parameter values as that in Fig.12 except $B1 = B2$. θ is varied from 10 to 100. In general, a higher value of θ is not favorable to both indexed methods. However, even when θ is 100, the SDB method is still much faster than the hybrid hash join.

These performance results show the performance advantage of the SDB method under

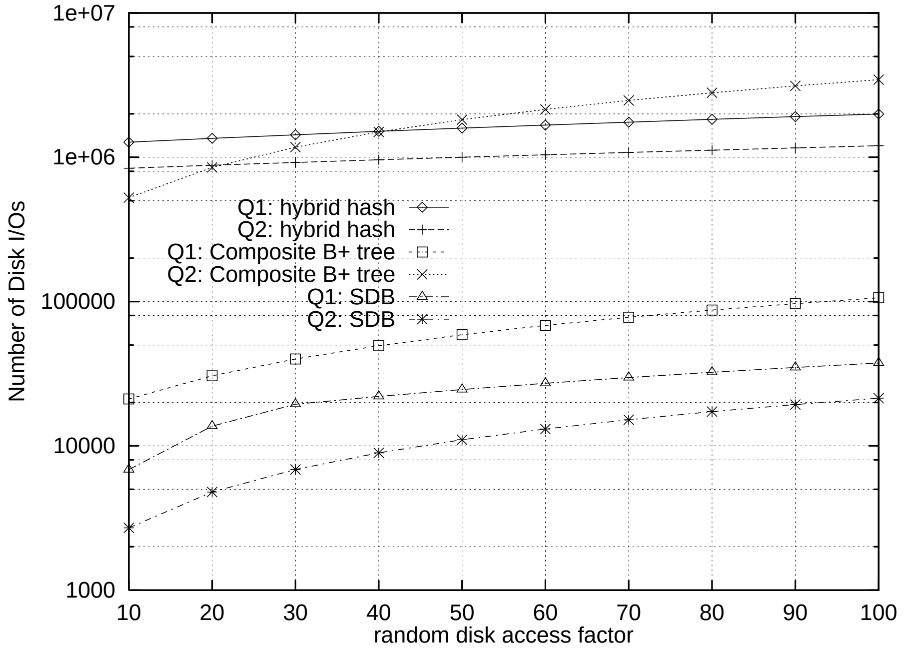

Figure 13: $\mid B1 \mid = \mid B2 \mid = 200k$ pages; $s = 10k$ pages; $\alpha = 0.0001$.

a wide spectrum of conditions, as compared to the B_c tree method and the hybrid hash join. The query execution of the SDB method is especially fast for complex queries having multiple join and/or selection predicates, though we used simple queries in the performance analysis for simplicity of formulas and ease of explanation.

7 Summary Remarks

In retrospect, a new indexing structure, SDB, is proposed in this paper. A SDB uses signatures to embody summary information for groups of values in the database. The novelty of the SDB method is that a small yet complete database, SDB, contains summary information and has a similar schema as that of the main database, is used to facilitate join as well as selection executions in the main database.

For join executions in very large databases, the SDB method is faster than the B_c tree method, which in turn is faster than the join index method. Join index has been shown to have an excellent performance in very large databases [8] with low join selectivities as compared to hybrid hash join and materialized views. So, the SDB method has an excellent performance in very large databases with low join selectivities.

The SDB method is not just a join method. A full set of relational operations, including joins and/or selections, can be executed in a SDB before the main database is even accessed. A SDB provides fast access for every attribute in the main database. Compared to the large storage cost of the approach of a fully inverted file for each relation in the main database, a SDB is a light-weighted access index which can provide comparable performance for selection executions.

The SDB method uses a data structure that is larger than that of the B_c tree method (which is much smaller than that in the join index method) while while providing a much better join execution performance than the B_c tree method when relations are large and the join selectivities are low.

As with any complex indexing scheme, a SDB 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.

The effectiveness of using the SDB method depends on the data distribution in the database. Our analysis shows that even in the worst case, the SDB method can significantly speed up query execution, especially join execution, when join selectivity is low and database is large.

Future databases are likely to be much larger than today's databases, and there is an increasing need of 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 joins in very large databases makes the SDB method attractive for such applications. Also, the excellent performance of the SDB method in answering complex queries in very large databases also makes the SDB method attractive as an access index for answering OLAP type queries in data warehouses.

Further improvement in the SDB method is possible. Because of the small size, summary relations and bonding tables may be stored entirely in the memory, resulting in an even faster performance. There are also some research issues in implementing the SDB method. For example, traditional query optimization strategy, in which selections are preferred before joins, fails to yield significant performance advantage when applied to the SDB method (and the B_c tree method as well) due to the small cost of executing a low selectivity join in a large database.

Appendix: Signature Method

A *signature* is a 0/1 bit pattern obtained by the hashing of data values. Fig.14 shows a relation with an attribute "profession". Each value in "profession" is hashed into a signature, called an *individual signature*. Next, individual signatures are grouped and superimposed to form a *group signature* (*sig1, sig2*, and *sig3* are group signatures). Consider the following query:

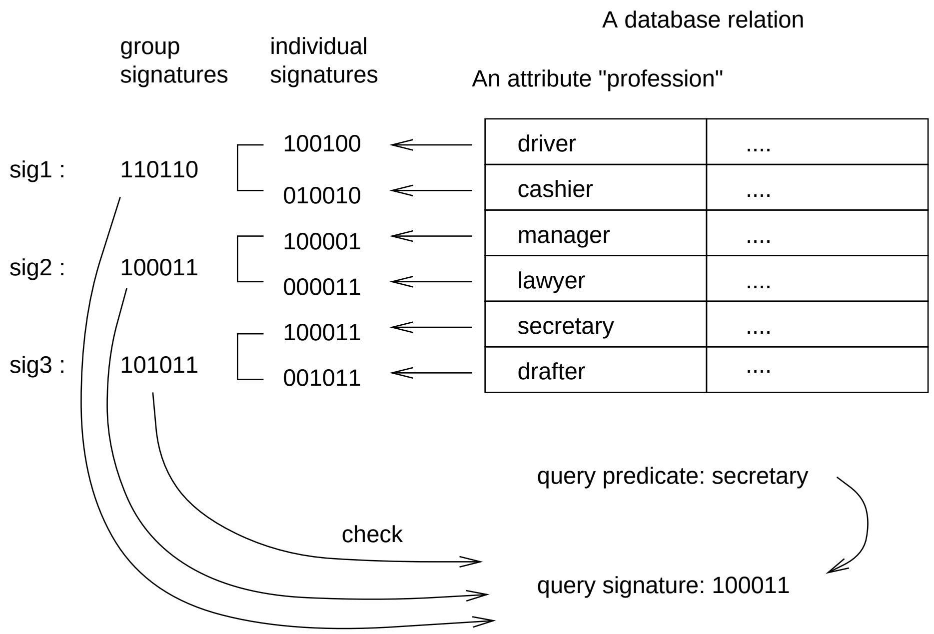

Figure 14: Individual signatures, group signatures and the query signature.

$$\begin{array}{ll} select & R.name \\ from & R \\ where & R.profession = \text{``}secretary\text{''} \end{array}$$

The value "*secretary*" in the predicate of the query is hashed into a signature $qsig$, called a *query signature*. $qsig$ is checked against group signatures. A *match* of signature $qsig$ with a group signature is defined to be that each bit of 1 in $qsig$ is matched by a bit of 1 in the same position of the group signature. A match under this definition implies that what a query is looking for might exist in data that is represented by the matching group signature. Note that under this definition, it could be the case that some 0's of $qsig$ may correspond to 1's in the same position of a matching group signature.

This definition of *match* can be reversed: each bit of 1 in a group signature is matched by a bit of 1 in the same position of $qsig$. In this case the query is a range (or set) query that asks if there is any value in the database that falls within this range. Thus, the definition of match can be altered according to the needs of applications. To keep things simple, we will stick to the first definition that a match of signatures means each bit of 1 in $qsig$ is matched by a bit of 1 in a group signature.

The group of values whose individual signatures are superimposed to form $sig1$ cannot possibly contain value "*secretary*" because $qsig$ does not match with $sig1$. Although $qsig$ matches with $sig2$, further checking shows that this is a false match. Only $sig3$ has a true match with $qsig$. If the group is large, for example, the average group size is 40, and a large number of group signatures do not match with $qsig$, then a significant amount of checking of the original data in the database can be avoided.

References

[1] S. B. Yao. "Approximating Block Accesses in Database Organizations," *Commun. ACM 20, 4, pp. 260-261, April 1977.*

[2] T. Haerder. "Implementing a Generalized Access Path Structure for a Relational Database System," *ACM Transactions on Database Systems, Vol.3, No.3, pp. 285-298, September 1978.*

[3] J. L. Pfaltz, W. J. Berman, E. M. Cagley. "Partial-match retrieval using indexed descriptors file," *Communications of ACM, 23, 9, pp. 522-528, Sept. 1980.*

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

[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] J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, Y. Wang. "An Efficient Hybrid Join Algorithm: A DB2 Prototype". *Proc. of 7th Int'l Conf. on Data Engineering, IEEE, pp. 171-180, 1991.*

[10] J. Han, Y. Cai and N. Cercone. "Concept-Based Data Classification in Relational Databases". *Workshop Notes of 1991 AAAI Workshop on Knowledge Discovery in Databases (KDD'91), Anaheim, CA, pp. 77-94, July 1991.*

[11] M. Negri and G. Pelagatti. "Distributive Join: A New Algorithm for Joining Relations". *ACM Transactions on Database Systems, Vol. 16, No. 4, pp. 655-689, December 1991.*

[12] J. Han, Y. Cai, and N. Cercone. "Knowledge Discovery in Databases: An Attribute-Oriented Approach". *Proc. of 18th Int'l Conf. on Very Large Data Bases, pp. 547-559, August 1992.*

[13] Z. Lin, C. Faloutsos. "Frame-Sliced Signature Files," *IEEE Transactions on Knowledge and Data Engineering, Vol. 4, No. 3, pp. 281-289, June 1992.*

[14] K. Smith, M. Winslett. Entity Modeling in the MLS Relational Model. *Proc. of the 18th Int'l Conf. on Very Large Data Bases, pp. 199-210, August 1992.*

[15] J. Han, Y. Cai, and N. Cercone. "Data-Driven Discovery of Quantitative Rules in Relational Databases". *IEEE Transactions on Knowledge and Data Engineering, Vol.5, pp. 29-40, February 1993.*

[16] Valery Soloviev. "A Truncating Hash Algorithm for Processing Band-Join Queries". *Proc. 9th Int'l Conf. on Data Engineering, IEEE, 1993.*

[17] J. Han, Y. Fu and R.T. Ng. "Cooperative Query Answering Using Multiple Layered Databases". *Proc. of the 2nd Int'l Conf. on Cooperative Information Systems, Toronto, Canada, pp. 47-58, May 1994.*

[18] J. Han, O. R. Zaiane, Y. Fu. "Resource and Knowledge Discovery in Global Information Systems: A Multiple Layered Database Approach". *Technical Report CSS/LCCR TR94-24, CMPT TR94-10, Database Systems Laboratory, School of Computing Science, Simon Fraser University, 1994.*

[19] J. M. Patel, M. J. Carey, M. K. Vernon, "Accurate Modeling of The Hybrid Hash Join Algorithm." *Performance Evaluation Review, Vol. 22, No. 1, pp. 56-66, May 1994.*

[20] X. Qian. "A Model-Theoretic Semantics of the Multilevel Relational Model". *Advances in Database Technology - EDBT'94, Lecture Notes in Computer Science 779, pp. 201-214, Jarke, M., Bubenko, J. and Jeffery, K. (editors), Springer-Verlag, 1994.*

[21] 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.*

[22] M.-L. Lo and C. V. Ravishankar. "Towards Eliminating Random I/O in Hash Joins". *Proc. of 12th Int'l Conf. on Data Engineering, IEEE, February 26-March 1, pp. 422-429, 1996.*