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

January 28, 1998

Abstract

This paper proposes a new summary database architecture (SDB) that employs summary information in the forms of signatures for text-valued data and statistics such as the average, minimum, and maximum for numerical-valued data in the execution of joins and selections in a large relational database. Its join execution performance is better than the B_c tree method [4]. Its performance advantage is especially evident in large database for queries with low join selectivity or complex queries that have multiple joins and/or selections. A distinct feature of the SDB method is that for a given query, a full set of relational operations, including joins and/or selections, can be executed in the SDB before the main database is accessed. A detailed comparison between SDB and B_c tree join execution performance is also included. The excellent performance in answering queries with low selectivity joins in large relational databases makes the SDB method ideal for various "online" databases such as the databases over the World Wide Web or in mobile computing environments.

Key words: Relational database, query execution, join, selection, equijoin, index, summary, disk I/O, 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 the relational database because it typically requires several passes over entire relations. Techniques for executing joins can be classified as: (1) non-indexed: the nested loop join, the

sort-merge join, and the hybrid hash join, etc.; (2) indexed: the nested block join with B+ index, the distributive join [9], the join index [3], and the composite B+ tree index [4], etc.

In a database environment, if the average size of join results is small, then using an index results in a few random disk accesses and thus can be much faster than scanning the entire relation. So, a large database with typically small join results favors the use of an indexed join method.

With the recent advancements in mobile computing and the World Wide Web, a new trend in the database industry is to set up databases that provide information about products and services directly to end customers. Since average end customers do not need inordinate amounts of information, these databases, though large, will have limited size of query results. It is apparent, then, that indexed join methods will perform better in databases in mobile computing environments and the World Wide Web.

The simplest indexes for speeding up join executions are indexes that are originally built for performing selections such as the B+ tree. For clarity, let us call these indexes the *selection indexes*. A selection index can be used to speed up join executions along with nested looping (nested block join with index), sorting [7], and hashing [9]. The join index [3] is one of the cross-relational indexes, specially designed for join executions. It is not widely used [4] in commercial implementation of relational database systems due largely to its excessive large disk storage requirement. The B_c tree [4] is another cross-relational index with a much smaller data structure. Except for the join index and the B_c tree method, all other join methods require minimum of one pass over at least one of the joining relations, which can be time-consuming in large relational databases.

The layered database concept has been proposed mostly for improving the modeling capability of the database or database security [11, 16]. Some have used multiple layers as a way of incorporating various databases in a heterogeneous system such as the World Wide Web [15]. A multi-layer approach was also proposed to speed up the query execution [14] However, due to difficulties in executing joins in upper layers, the join execution was not done in the upper layer [14].

This paper proposes a new summary database architecture (SDB) that can be used to speed up joins as well as selections in large relational databases. The major components of 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, 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, if the query contains joins or multiple relational operations. The result of the query execution is mapped by addressing table to base relations. Because summary relations are much smaller than their base relations, the query

execution in the summary relations is much faster.

The query result from the summary relations is mapped by addressing tables to a set of join attribute values and tuple pointers (TIDs). Addressing tables are implemented by B+ trees. So, to be more specific, the query result from the summary relations is used to selectively load disk pages containing leaf nodes of the B+ trees into the memory. These loaded pages of leaf nodes contain join attribute values and TIDs relevant to this query. If the query contains join operations, the joins are further executed using these loaded leaf nodes of the addressing tables in the memory.

If the query contains only joins, then the query execution result from the SDB is a set of TIDs which point to precisely the set of tuples (in the main database) which constitute the query result. If the query contains joins and selections, or only selections, then the query execution result from the SDB is a set of TIDs which point to a set of tuples (in the main database) which is a superset of the true query result. After the query execution in SDB, disk pages containing relevant tuples (in the main database) are loaded to finally finish off the execution for the query. The difference between this "after SDB" query execution and a query execution without using a SDB is that this "after SDB" query execution works on a much reduced data set, and thus is much faster.

Section 2 introduces the proposed summary database structure. Section 3 presents a set of guidelines for grouping tuples in constructing the summary database. Section 2 centers around text-valued data whose summary information is embodied in signatures. Section 4 discusses numerical data and using statistics to embody the summary information. Section 5 compares our approach with other join methods. Section 6 gives a performance comparison of our approach to the B_c tree and the hybrid hash join methods. Finally, Section 7 concludes the paper.

2 The Summary Database (SDB)

In the join index method, the size of join indexes is huge, and a scan over a join index may be no less expensive than a scan over the joining relations themselves. The B_c tree method can execute joins faster than the join index because it uses a smaller data structure [4] and B+ trees. This paper proposes a *summary database* (SDB) method which is a further improvement over the B_c tree method. In fact, the SDB method is more than just a join execution method. A SDB is an index structure which can assist in join executions as well as selections. The SDB is like a small database such that queries with multiple joins and/or selections can be executed on it.

A SDB is constructed from the 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

each relation in the main database, if there are j join attributes[1], then j relations (*summary relations*) will be built in the SDB for that main database relation. The main database relation is called the *base* relation of these summary relations. Each summary relation has exactly the same schema, i.e., the same set of attributes, as its base relation. A summary relation is made by grouping tuples in its base relation. Each tuple in a summary relation, i.e., *a summary tuple*, is made by extracting summary information from a group (*the base group*) of tuples, i.e., *base tuples*, in the base relation. Each entry (which corresponds to an attribute) in a summary tuple contains a summary information from the same attribute of its base tuples. The grouping of tuples in the base relation is based on the grouping of attribute values in the base relation.

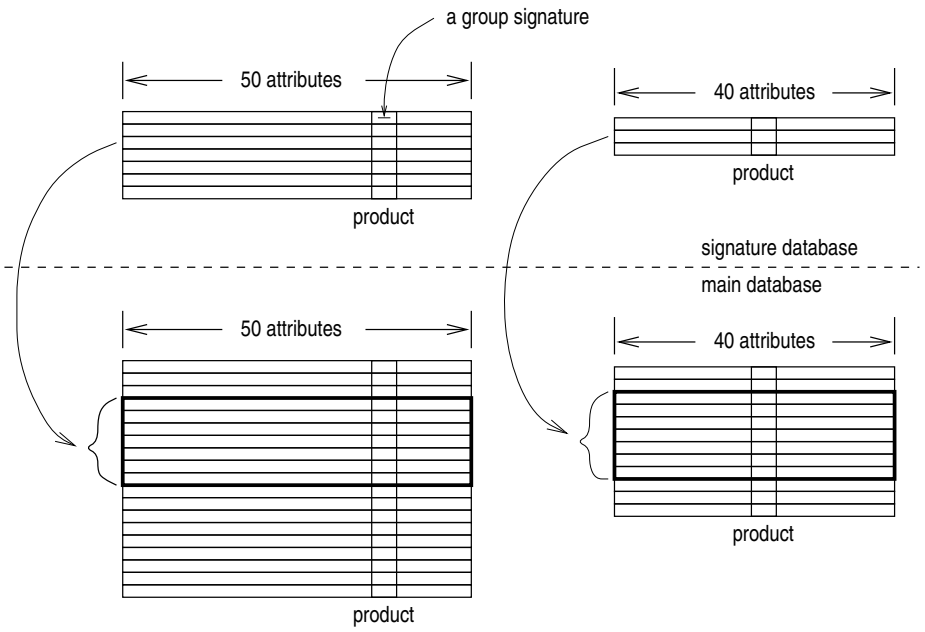

Figure 1: Correspondence between base relations and summary relations.

For example, if a base relation has 50 attributes and three of them are join attributes, then there are three summary relations for this base relation and each of the three summary relations have 50 attributes. An entry corresponding to an attribute in a summary tuple of a summary relation contains summary information of that attribute from its base tuples in the base relation, as shown in Fig. 1.

The summary information can take a variety of forms. If an attribute has a text-valued data domain, then each attribute value (in this attribute) in the base relation is hashed into an individual signature. A group signature is obtained by superimposing a group of individual signatures from the same attribute in the base group of the base relation and this

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

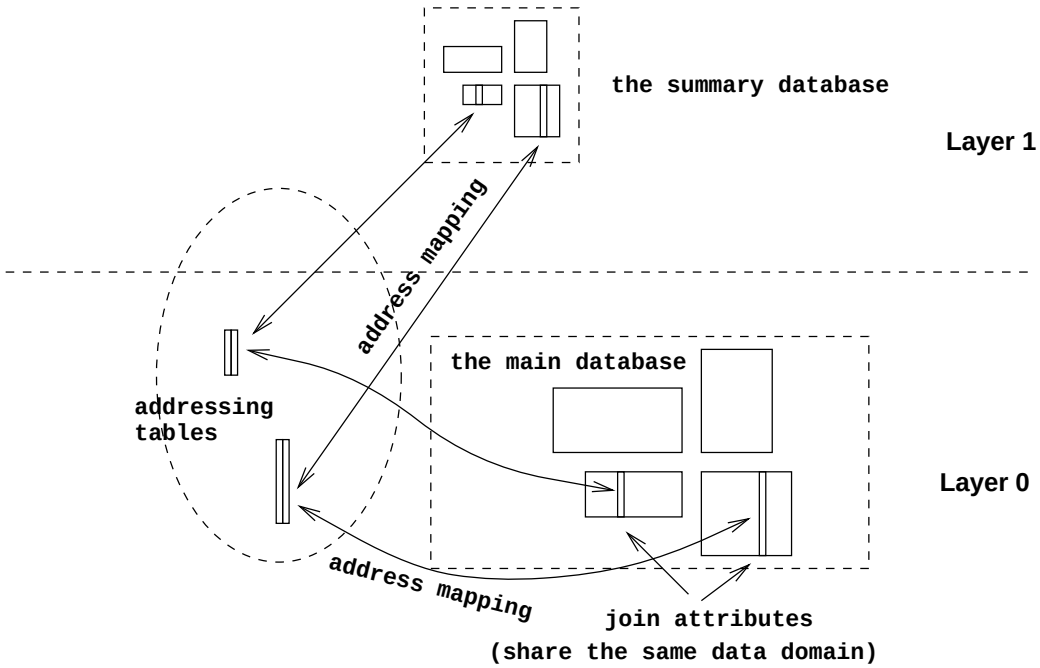

Figure 2: The architecture of a SDB along with its main database.

group signature is used as a summary information.

Fig. 2 shows the overall structure of a SDB along with its main database.

Query execution starts from the SDB. For text data, signatures are used to embody the 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. We had given an overall description of the query execution using SDB in Section 1 and it will become more clear later in this section when we discuss bonding and addressing tables.

The size of the SDB can be made small by choosing proper group size and grouping criteria of tuples in the main database, and will be discussed in detail in Section 3.

For simplicity of discussion, this section focus on text data (using signatures to embody summary information). Numerical data (using statistics to embody summary information) will be discussed in Section 4.

Construction of a SDB: Let A be a join attribute with a data domain D in a relation R. Let S_D be the set of distinct values in all attributes in 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 UID.

From each partition in S_D, attribute A of R takes its own share of values from the partition and these values form a group. This group of attribute values corresponds to a group of tuples in R. A signature is hashed out of each entry of each tuple in the group (*the base group*). For each attribute (all attributes, not only join attributes. See Fig. 1) 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. This summary tuple bears the same UID as the corresponding partition in S_D. We emphasize once again that though partitioning of tuples in the main database is done based on the partitioning of values in the join attribute, group group signatures are made from all attributes in the base relation. All summary tuples constructed based on a partitioning in values of attribute A form a summary relation.

This process is repeated for every join attribute in the main database that have the same data domain D. All summary tuples made out of the same partition of S_D bear the same UID, though these summary tuples can reside in different summary relations of different base relations. These UIDs logically connect 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 and construction of group signatures. Let A be a join attribute that has D as its data domain and let $S_D(A)$ be the set of all values in A. After S_D is partitioned, $S_D(A) \cap P$ 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 $i \neq j$; $J = \{j_1, j_2, ..., j_s\}$ be the set of all join attributes that have data domain D; S_t be the set of values in j_t, $1 \leq t \leq s$ (Note that $\cup_{t=1,2,...,s}\ S_t = S_D \subseteq D$). Then partitioning in each S_t $(1 \leq t \leq s)$ can be done as $PV = \{\ S_t \cap p_1,\ S_t \cap p_2,\ ...\ ,\ S_t \cap p_s\}$.

Bonding tables: A bonding table is used to accomplish join execution between summary relations as well as to propagate query result from one summary relation to another if they have the same base relation. Bonding tables are constructed between every pair of summary relations based on different yet joinable base relations (shown in Fig. 3) and between every pair of summary relations based on the partitioning of different join attributes of the same base relation.

In Fig. 3, 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 $B2$ are joinable through join attributes B and C. Horizontal line segments on the base relations $B1$ and $B2$ in Fig. 3 represent groupings of tuples in base relations. Each summary tuple in S_A, S_B or S_C has a UID and is a tuple of group signatures.

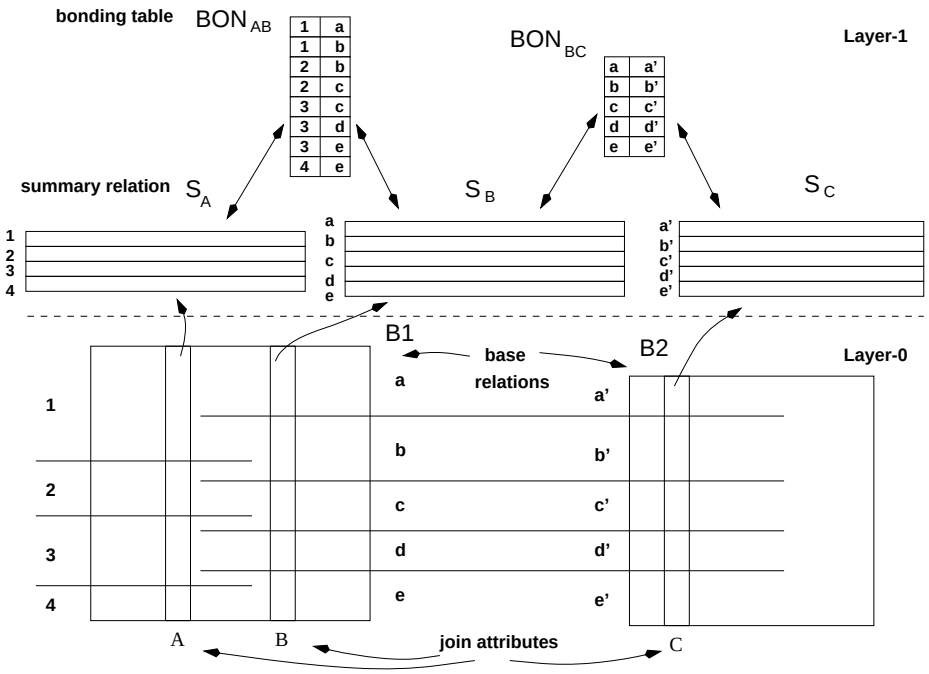

Figure 3: Bonding tables and summary relations.

Every tuple of a bonding table has two components, each is a logical address of a summary tuple in one of the two summary relations that are connected by this table. Two tuples linked by the bonding table in this manner bear the same UID. In Fig. 3, tuple 2 in summary relation S_A corresponds to tuples b and c in summary relation S_B, and this is recorded in the bonding table BON_{AB}.

Let us take the following query example in conjunction with Fig. 3 to illustrate a join execution in summary relations using bonding tables:

$$
\begin{array}{ll}
select & B1.name \\
from & B1,\ B2 \\
where & B1.B = B2.Cand \\
 & B2.category = \text{“}autoparts\text{”}.
\end{array}
$$

If “autoparts” reduce qualified summary tuples in S_C to b’ and c’, then through bonding table BON_{BC}, the join result in the summary relation can be determined as {bb’, cc’}, in which the juxtaposition of symbols represents the concatenation of tuples. Note the meaning of a join in the summary relations: a joined tuple bb’ indicates that some or all of the tuples in the base group of summary tuple b in $B1$ may be joinable with some or all of the tuples in the base group of b’ in $B2$. Through a join between S_B and S_C, all tuples in $B1$ and $B2$ that are in the base groups of summary tuples a, d, e, a’, d’, and e’ can be excluded from the join execution. This greatly reduces the cost of further query processing in the main

database.

Bonding table BON_{AB} links two summary relations that share the same base relation. To illustrate the functionality of BON_{AB}, let us suppose that above query had a join involving attribute A and another attribute (not shown in Fig. 3). In the execution of that join, suppose that summary tuples 3 and 4 in the summary relation S_A were found to be relevant to the query. The result of the additional join and {bb', cc'} would be further joined by using the bonding table BON_{AB} to obtain the final query result { 3cc' } from the summary relations.

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

If a query only contains joins, then only bonding tables are used in the query execution and signatures in the summary relations are not used. Only when a query contains selections are query signatures used to compare with signatures in summary relations for locating summary tuples that are relevant to the query (see Appendix A). 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.

It should be understood that the construction of summary relations and bonding tables is not done at the time of a query execution. Once the SDB has been been built, any number of queries can be executed without 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.

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 [4]. One may have similar concern about the size of bonding tables. However, unlike the join index [3], bonding tables are built between summary relations which are much smaller than main database relations (if the average size of each group is 100, then the size of a summary relation is only 1% of the size of its base relation). In addition, partitioning is an operation that can be subjectively 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 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 components: 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. After a query is

executed in the SDB, an addressing table is used to map resulting summary tuples to their corresponding base tuples.

A B_c tree [4] can be conveniently used as a part of an addressing table with logical addresses of summary tuples residing in non-leaf nodes and logical addresses of base tuples residing in leaf nodes. A B_c tree is a modified B+ tree; each leaf-node contains a value from a data domain and several TID (tuple pointer) lists. Each TID is a logical address of a tuple that has this value as its join attribute value and all TIDs in a list point to tuples in the same relation. Each data domain (of join attributes) has one B_c tree. A B_c tree can be used to do fast join executions since a B_c tree is much smaller than joining relations [4].

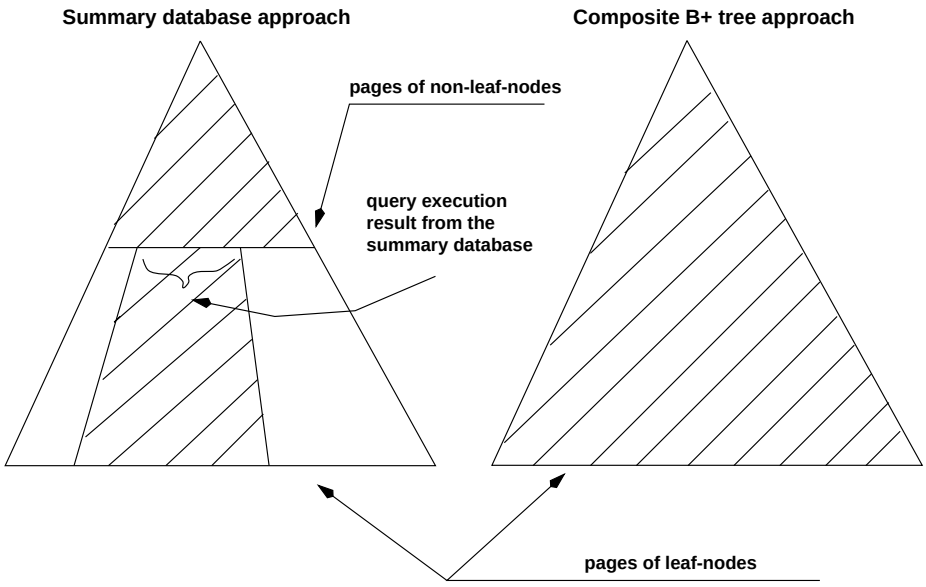

Figure 4: Search scope reduction of an addressing table in a data domain.

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 serves to reduce the search effort to find the query result and further processing is needed to determine the exact set of tuples of the result. For each data domain that is referred in a query, addresses of some summary tuples from 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 this data domain into the memory. This leads to a further reduction in the number of disk I/Os needed as compared to the B_c method, especially for a full join (nonselective join, i.e., all join attribute values are possible candidates of matching values [4]). Not even all the pages of leaf nodes of the B_c are 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 100, then the maximum number of relevant leaf nodes, as well as the maximum number of pages that contains these leaf nodes (assuming

one leaf-node reside in one page. This is true because usually a disk page can contain a large number of leaf nodes), is $100 * j_s$, which can be much less than the number of all pages containing leaf nodes of the B_c tree for a very large database. This is shown in Fig. 4.

Summary relations and bonding and addressing tables constitute a summary database (SDB). It should be clear that only the join execution can be executed completely in the SDB. For example, if a query contains (JS), where J is a join and S is a selection, 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 selection 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, another form of generalization is also possible: SDB can be duplicated, except addressing tables (they are much larger than the rest of the SDB), within the same layer to allow multiple queries to be executed in parallel in the same layer. Each query is executed sequentially while parallel executions are among multiple query executions.

A distinct advantage of this type of parallel join execution is that a full partitioning of the main database is not needed. Since most parallel join methods require partitioning and distribution of the data among different sites, our version of using multiple SDBs to achieve a parallel join execution represents a much simpler alternative.

Heuristic bonding tables: In some cases, such as online databases directly serving customer queries for product and service information, an exhaustive search for query result may not be necessary and users may be satisfied if sufficient (not necessarily all) information is returned from 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 is much more desirable than the completeness of query results.

This situation opens the door to further speed up query executions in SDB. 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 links in bonding tables.

In Fig. 3, 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 results 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 the 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 may be used.

Cost of constructing the SDB: The cost of constructing the SDB is independent of the cost of query executions because they occur at different times. After the SDB has been constructed, queries can be executed in the SDB without incurring any cost towards "construction of the SDB" as long as values in the main database are not updated.

The cost of constructing the SDB, in terms of computer time, measured by the number of disk I/Os, is a few scans over the entire database. If the database is not frequently updated (frequent database update is bad for any indexed method, not just bad for the SDB method), the cost of constructing a SDB should not be a problem.

3 Cascaded Partitioning Algorithm

Different grouping of tuples in the main database can result in different degree of interconnecting complexity between summary relations and this in turn result in different sizes 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.

Partitioning of tuples in various relations are related. Let $R1$ and $R2$ be two joinable relations. $R1$ and $R2$ can be partitioned independently. A better way is to partition only one of the relations first, for example, $R1$. Since $R1$ and $R2$ are joinable, the partitioning of $R1$ naturally induces a partitioning of $R2$. If the induced partitioning of $R2$ is not even enough, or some partitions of $R2$ are too large, we can further partition $R2$ based on the induced partitioning, i.e., to refine the induced partitioning in $R2$. The refinement of induced partitioning in $R2$ does not affect the partitioning in $R1$ that have already been done. Let the summary relations coming out of this partitioning be denoted as S_{R1} and S_{R2}. There is only one-to-many relationship between the summary tuples in S_{R1} and the summary tuples in S_{R2}. This means that the bonding table linking S_{R1} and S_{R2} has no more than $\mid S_{R2} \mid$ rows.

Hence, this partitioning algorithm can effectively 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?" 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. This type of common sense leads to a ranking of all join attributes in the main database based on the following five principles:

- P1: more important attributes are ranked higher. For example, in a supermarket database, price information may be ranked higher than color, contact phone numbers, etc.
- P2: more frequently used attributes are ranked higher.
- P3: attributes with more orderly data domain are ranked higher. For example, price of goods (a numerically ordered data) may be ranked higher than the description of goods.
- P4: more structured attributes are ranked higher. For example, the address of a customer (can be geographically grouped) may be ranked higher than the name of the customer.
- P5: 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 grouping of tuples in main database relations. Conflicts arise mainly because a main database relation can 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 a 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.

These principles can be further explained by the order of partitioning among different join attributes: join attributes with higher priorities should be partitioned ahead of join attributes with lower priorities. To be more specific, join attributes in larger relations (P5), more important join attributes (P1), more useful ones (P2), or more easily partitioned ones (P3 and P4) are assigned higher priorities. We impose no strict order among these principles. These principles are heuristics and may themselves be in conflict with each other when applied in practice. Should that happen, common sense and business knowledge should be applied in solving 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 one of them as the join attribute to be partitioned first. To determine the order of partitioning among a large number of join attributes, we need the help of a concept called linear ordering which is introduced below.

Linear ordering: A *linear ordering LO* is defined as a connected, directed graph with no cycles. Each node of the graph 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 LO. 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 consisting of join attributes A_1, A_2, ... , A_k with decreasing ranks can be denoted as $LO = (A_1, A_2, ..., A_k)$.

In general, it is possible to have several linear orderings co-exist in the main database. Both the way to derive linear ordering 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, there is no unique way to resolve the conflicts (these conflicts are resolved by applying common sense and business knowledge of the database designers).

Cascaded partitioning algorithm: Partitioning of join attribute values is done starting from the highest ranked 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.

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 way to evenly partition a set of values [13]. In order to evenly partition 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.

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 D_1 ($S_{A_1} \subseteq D_1$). 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 D_2 (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 par-

titioning 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 $pp'_i = \{t[A_2] \mid t[A_1] \in pp_i\}$, $1 \leq i \leq k$. Then, PS_{A_2} can be further refined, as discussed in the beginning of this section, to become a more even partitioning.

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_{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's, d_i's are mutually non-interset.

This process is repeated for each node of LO. Each linear ordering is processed 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, and the average number of tuples in a bonding table is even lower. Since summary relations are very small, so are the bonding tables link them.

The partitioning of values in join attributes can also be helped by using concept hierarchies. Concept hierarchy [8, 10, 12, 14] 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.

The next section discusses ordered data domains. An ordered data domain is one in which order between data items makes sense, such as a numerical data domain or a string data domain with a lexicographical ordering. Note that object IDs such as employee ID, product ID, are not considered ordered data. This type of "ID" attribute is frequently used in commercial implementation of relational databases as the key attribute of relations, and often it is a join attribute. "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 Attributes with Ordered Data Domains

For the simplicity of discussion, only one-dimensional (1D) or high dimensional (HD) numerical data domains are discussed in this section. The main difference in query execution

between text-valued and numerical-valued attributes is that numerical valued attributes are often involved in range queries. Instead of group signatures, statistics such as the average, minimum, and maximum values are used to compactly represent a group of numerical values. Partitioning by range or clustering such as that used in the BIRCH project [18] can be employed to group numerical values. Though there are many choices of summary information of numerical values, any useful choice must be able to capture some essential characteristics of a group of numerical values (1D or HD): the center of the group and the defining boundary of the group. For example, $v = (avg, min, max, n)$ can be a choice of a summary information in which avg represents the center of the group, (min, max) represents the defining boundary of the group, and n is a count of the number of base tuples. Cascaded partitioning algorithm described in Section 3 can be directly used for the partitioning of numerically valued attributes.

Convex property: In one-dimensional space, a convex partitioning is one that between any pair of values within the same group, there lies no values of any other group. It is obvious that convex property is desirable for partitioning of values (such as in the construction of SDB) in preparation for answering range queries such as "looking for all products from supplies with prices within \$1 of the asked prices from the supermarket". The concept of convex can be generalized to higher dimensional spaces and is a useful concept in partitioning of values in preparation for answering range queries in higher dimensional spaces.

Formerly, let S be a Euclidean space; $PS = \{G_1, G_2, ..., G_k\}$ be a partitioning of S, i.e., $S = \cup_{i=1,2,...,k} G_i$ and $G_i \cap G_j = \emptyset$, if $i \neq j$. We say PS is convex, if $vj \notin \{t*vi_1+(1-t)*vi_2 \mid 0 \leq t \leq 1, vi_1, vi_2 \in G_i\}$, $\forall vj \in G_j$, $\forall i \neq j$.

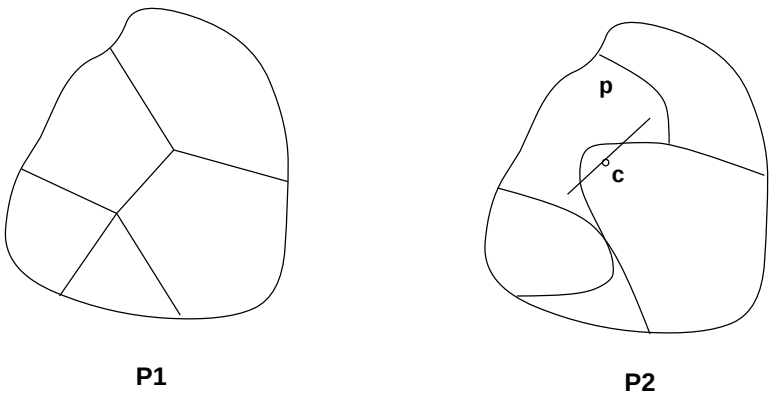

Figure 5: Illustration of convex property in 2D space.

Partitioning $P1$ in Fig. 5 shows a convex partitioning in a two-dimensional space. Partitioning $P2$ in Fig. 5 is not a convex partitioning because c lies outside a partition p. If this is about the partitioning of a geographical area, in most cases, partitioning $P1$ is more

desirable than $P2$.

Query execution using summary information: A query execution in the summary database using numerical summary information (NSI) entries is similar to the case of text-valued data. A match between join attribute NSI entries v_1 and v_2 of two summary tuples t_1 and t_2 means that in the main database, some of the base tuples of t_1 may be joinable to some of the base tuples of t_2. Because NSI entries such as (avg, min, max, n) represent ranges of values, $v_1 = (avg_1, min_1, max_1, n_1)$ and $v_2 = (avg_2, min_2, max_2, n_2)$ are considered to be matched if their ranges overlap, i.e., $[min_1, max_1] \cap [min_2, max_2] \neq \emptyset$.

Updating of NSI entries: To compute a new average value v'_{avg} after modifying a value $a' \in G = \{a_1, a_2, ..., a_n\}$ is straightforward. If a value v is inserted into G which is represented by summary information $GI = (avg, min, max, n)$, v'_{avg} can be computed as:

$$\begin{aligned} v'_{avg} &= \frac{a_1+a_2+...+a_n+v}{n+1} \\ &= \frac{a_1+a_2+...+a_n}{n} * \frac{n}{n+1} + \frac{v}{n+1} \\ &= \frac{avg*n+v}{n+1} \end{aligned}$$

If a value $a' \in G$ is deleted from G, the new average value can be computed as:

$$\begin{aligned} v'_{avg} &= \frac{a_1+a_2+...+a_n-a'}{n-1} \\ &= \frac{a_1+a_2+...+a_n}{n} * \frac{n}{n-1} - \frac{a'}{n-1} \\ &= \frac{avg*n-a'}{n-1} \end{aligned}$$

5 Relations to Previous Techniques

Indexes (e.g., B+ tree) that were originally designed for selections can be used to speed up join executions [7, 9]. 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 [3], and B_c tree [4].

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, where m is the number of relations in the database; while 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; while 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. Since all leaf nodes constitute the bulk of a B_c, the size of a B_c tree is about $(n_1 + n_2) * TID_{sz}$, where TID_{sz} is the size of a TID (logical address of a tuple).

To appreciate the size difference of the data structures used by the B_c tree method and that used by the join index method, we can estimate them in order of numbers: the number of indexing structures in 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 $O(n_1 + n_2)$.

The SDB method is a further improvement over the B_c tree method. As a small database itself, a SDB can execute a query that contains multiple joins and/or selections, whereas a B_c is just an index for the join execution.

Joins can be executed faster by using the SDB method than the B_c method in two cases: (1) when joining relations are large and join selectivity is low; (2) when a 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 very 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. We present a numerical example to explain this fact. Assume each joining relation has 10^5 tuples; the join result has 100 tuples, joined from 100 tuples from each joining relation; 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 the join execution from the summary relations, the number of summary tuples from each joining summary relation cannot exceed 100, because each pair of matching summary tuples must contain at least one pair (as the worst case scenario) of matching base tuples in the main database. 100 summary tuples can be mapped to at most $100 \times 20 = 2,000$ base tuples in each of the joining relations; so only $2,000$ TIDs of each relation need to be checked for matches. This represents a reduction of $(10^5 - 2,000)/10^5 = 98\%$.

The execution of each join or selection in the summary relations (and the bonding tables) represents a possible reduction of the query result from the summary relations (and the bonding tables). So, it is clear that the more joins and/or selections a query contains, the more reduction in the size of query result can be achieved in the SDB. So, the SDB is also better suited to answer complex queries with multiple joins and/or selections.

The size of all data structures in the SDB method is slightly larger than the size of all data structures in the B_c tree method. The size of all addressing tables, which constitute the bulk of all data structures in the SDB method, is about the same as the size of all data structures in the B_c tree method, which is of the order of $O(mn)$ (sse previous discussion in this section). In addition, the SDB has $O(m)$ summary relations and bonding tables.

The size of each summary relation or bonding table is of the order of $O(n/g)$, where g is the average group size used in the construction of the SDB. So, roughly, the ratio of the size of all data structures in the SDB method to the size of all data structures in the B_c tree method is $1 + 1/g$, obtained by $(mn + mn/g)/(mn)$.

Although signatures are used to embody summary information of group of text values due to the convenience of using the signatures to superimpose them and obtain summary information, the SDB is far from a simple application of the signature method. In fact, it is better to use statistics (not signatures) such as the average, minimum, and maximum to represent summary information in a SDB when dealing with numerically valued attributes (see Section 4), especially when range queries are anticipated.

Also, the performance advantage of join executions in a SDB (though SDB is more than just a join method) as compared to other join methods is evident from the results of other studies. For example, a performance comparison among the hybrid hash join, materialized view, and join index [6] 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. The B_c tree method is an improvement over the join index [4]. As discussed previously in this section, the SDB method performs even better than the B_c tree method when join selectivity is low and joining relations are very large.

As for selections, building an index, e.g., a B+ tree, for each attribute of each relation is a very expensive approach in terms of the disk storage space. The SDB can serve as a light-weighted index in case indexes on all attributes in the database are desired.

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 the last section, it is clear that the B_c tree method is the closest join method to the SDB method, though 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 hybrid hash join, which is one of the best non-indexed methods. The comparison of these join methods is done under a common parameter setting. For performance comparison, we use the following two queries:

$Q1$:

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

and

$P_{sz} = 2,048$	page size (in bytes) of disk pages
$M = 5,000$	size (in # pages) of available memory space
$\mid B1 \mid = 100k$	# pages of $B1$
$\mid\mid B1 \mid\mid$	# tuples of $B1$
α	$\alpha\%$ tuples of $B1$ are joinable to tuples in $B2$
$\beta = 3$	each joinable tuple of $B1$ can join β tuples in $B2$
$g = 100$	grouping size (for constructing the SDB)
$b1_0$, $b2_0$	size (in pages) of anchor buckets of $B1$ and $B2$
$seq = \mid buf_{bucket} \mid$	$seq = \mid buf_{bucket} \mid$ 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: $\theta = t_{random}/t_{sequential}$
$TID_{sz} = 32$	the size (in bytes) of a TID
$A = 32$	the average size (in bytes) of data entries

Table 1: Parameters used in the performance comparison.

$Q2$:

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

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.

Assumptions: Performance is measured by the number of equivalent sequential disk I/Os (the time cost of a sequential disk access is 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 1. The cost of the writing out of join results is not included in the comparison because 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 [6]:

$$Y_{io}(k,m,n) = Yao(k,m,n) + Yao(k,n/FO,n) + T3(k,n), \tag{1}$$

where

$$T3(k,n) = Yao(Yao(k,n/FO,n),n/(FO*FO),n/FO). \tag{2}$$

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

$$Yao(k,m,n) = m - m * \prod_{i=1}^{k} \frac{n-(n/m)-i+1}{n-i+1}. \tag{3}$$

Query execution cost in the hybrid hash join: Some disk accesses of the hybrid hash join are 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 somewhat 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 [17]. 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 reduced to $\theta_{bucket} = (\theta + 50 - 1)/50 = 2.98$. If there are total of 20 hash buckets, then $100k \times 20 = 2MB$ memory space is used for these buffers and 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:

$$\begin{aligned} C_{hybrid\ \ hash(Q1)} = \ & \mid B1 \mid + max\{0,(\theta_{buckets}+1)(\mid B1 \mid - b1_0)\} \\ & + \mid B2 \mid + max\{0,(\theta_{buckets}+1)(\mid B2 \mid - b2_0)\} \end{aligned} \tag{4}$$

The first two terms represent the time to read pages of $B1$ into the memory for hashing, flushing out (note the random access factor $\theta_{buckets}$) filled buffers of 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 $0.9*M$ 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 selection results back to the disk) disk I/Os but the join cost reduces to:

$$\begin{aligned} C_{hybrid\ \ hash(Q2)} = \ & \mid B1 \mid + max\{0,(\theta_{buckets}+1)(\mid B1 \mid - b1_0)\} \\ & + s + max\{0,(\theta_{buckets}+1)(s - b2_0)\} \end{aligned} \tag{5}$$

Query execution cost in B_c: For query Q1, a full scan over all leaf nodes of a B_c tree is needed [4]. The disk I/O cost of such an execution is:

$$\begin{aligned} C_{B_c(Q1)} = \quad & (\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz} \\ & + \theta * Y_{io}(\alpha \| B1 \|, \mid B1 \mid, \| B1 \|) \\ & + \theta * Y_{io}(\alpha\beta \| B1 \|, \mid B2 \mid, \| B2 \|) \end{aligned} \tag{6}$$

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 joining 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 [4]. The cost of the join execution in Q2 is:

$$\begin{aligned} C_{B_c(Q2)} = \quad & \theta * Y_{io}(min\{5 * s, \| B1 \|\}, \\ & (\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz}, \| B1 \| + \| B2 \|) \\ & + \theta * Y_{io}(\alpha \| B1 \|, \mid B1 \mid, \| B1 \|) \\ & + \theta * Y_{io}(\alpha\beta \| B1 \|, s, 5 * s) \end{aligned} \tag{7}$$

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

Query execution cost of using the SDB: Performance of the join execution using the SDB method depends on the join selectivity in the summary relations. More precisely, the extra time saved with the SDB method over the B_c method depends on the join selectivity in the summary relations. The worst case scenario is no pruning at all: the result returned from the join execution in the summary relations indicates that all tuples in joining relations are candidates of joinable tuples. However, as discussed in Section 5, if the selectivity is low enough, this worst case will not happen, i.e., pruning is bound to happen. For query Q1, the disk I/O cost for a join execution in a database equipped with a SDB is:

$$\begin{aligned} C_{SDB(Q1)} = \quad & (\mid B1 \mid + \mid B2 \mid)/g + \mid B2 \mid /g \\ & + \theta * Y_{io}(min\{j_s * g, \alpha\beta \| B1 \| * g, \| B1 \|\}, \\ & min\{j_s * g, (\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz}\}, \\ & min\{j_s * g * 5, \| B1 \| + \| B2 \|\}) \\ & + \theta * Y_{io}(\alpha \| B1 \|, \mid B1 \mid, \| B1 \|) \\ & + \theta * Y_{io}(\alpha\beta \| B1 \|, \mid B2 \mid, \| B2 \|) \end{aligned} \tag{8}$$

The first term is the time to read summary relations. The actual time should be smaller because only the summary relation of $B1$ needs to be fully scanned, and through the use of a

bonding table, only selective entries in the summary relation of $B2$ are read. The second term is the time to read in a bonding table to execute the join in the SDB. Based on discussions in Section 3, a bonding table BON can be constructed such that $\| BON \| \leq \| B2 \| / g$. The third term denotes the time of selectively loading pages containing leaf nodes of an addressing table (its size is about the same as a B_c tree) into the memory to execute the join. This time depends on the join selectivity j_s in summary relations. It should be clear from discussions in Section 5 that j_s satisfies $j_s \leq \alpha\beta \| B1 \|$. So, if the join selectivity in the main database is low, only a part of $(\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz}$ pages are loaded into the memory. Note that if $j_s * g$ leaf nodes are to be accessed, at most $j_s * g$ disk pages are loaded into the memory since each disk page can contain a large number of leaf nodes, and at most $5 * j_s * g$ leaf nodes are contained in the loaded disk pages; hence, we obtain the second and third arguments in $Y_{io}(*)$ of the third term as the worst case scenario. The fourth and fifth terms denote the time to read joining tuples.

For query Q2, no additional execution cost occurs in the SDB because both summary relations are scanned and selection in the SDB can be done along with the scanning. The selection is done after the join execution is completed so that the selection is done on the join result. The selection costs the following additional disk I/Os:

$$\begin{aligned} C_{SDB(select)} = \quad & \theta * Y_{io}(min\{j_s * s_s * g, \alpha\beta \| B1 \| * s_m, 5 * s\}, \\ & \alpha\beta \mid B1 \mid * s_m, \alpha\beta \| B1 \| * s_m) \end{aligned} \tag{9}$$

Note that all values of parameters in the formula are reduced by a factor of s_m due to the selection in the SDB. The cost of the join execution in Q2 is reduced to:

$$\begin{aligned} C_{SDB(Q2)} = \quad & (\mid B1 \mid + \mid B2 \mid)/g + \mid B2 \mid / g \\ & + \theta * Y_{io}(leaves, \\ & min\{leaves, (\| B1 \| + \| B2 \|) * (A + TID_{sz})/P_{sz}\}, \\ & min\{leaves * 5, \| B1 \| + \| B2 \|\}) \\ & + \theta * Y_{io}(\alpha \| B1 \| * s_m, \mid B1 \mid, \| B1 \|) \\ & + \theta * Y_{io}(\alpha\beta \| B1 \| * s_m, \mid B2 \mid, \| B2 \|) \end{aligned} \tag{10}$$

where

$$leaves = min\{j_s * s_s * g, \alpha\beta \| B1 \| * g, \| B1 \|\} \tag{11}$$

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. 6, reffig:data2, and 8. The value of input parameters is shown in Table 6. The worst case performance of the SDB method is plotted in all figures. For example, if $\mid B1 \mid = 100K$ pages and the $\alpha = 0.0001$, then the join result should contain 50 tuples from $B1$, and the

number of summary tuples (in $B1$'s summary relation) resulted from the query execution in summary relations should be no more than 50; so $j_s = 50$ represents the worst case scenario (out of each 100 base tuples of a matching summary tuple, only one base tuple is joinable).

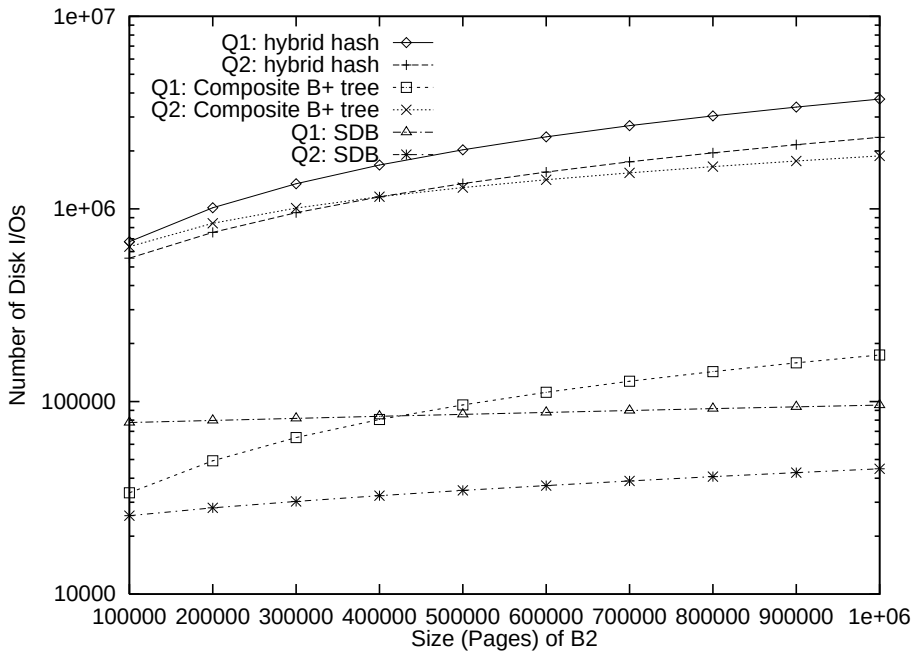

Figure 6: $\mid B1 \mid = 100k$ pages; $s = 5k$ pages; $\alpha = 0.0001$; $\theta = 20$.

In Fig. 6 and Fig. 7, the size of $B2$ varies from $\mid B1 \mid$ to $10 \mid B1 \mid$. The difference between Fig. 6 and Fig. 7 is that different join selectivities are used: $\alpha = 0.0001$ in Fig. 6 and $\alpha = 0.00002$ in Fig. 7. Note that relative selectivities $s_s = 20\%$ and $s_m = 80\%$ are conservative estimates because $s = 5,000$ means that the relative selectivity in $B2$ is less or equal to 5%.

From Fig. 6 and Fig. 7, we observe that the B_c tree method is competitive for executing a join without the selection (Q1), and is far less competitive for executing a join with a selection (Q2). This observation supports the finding in [4]. We also 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 the indexed methods, B_c and SDB, perform better in Fig. 7 than in Fig. 6 because of lower join selectivity. Our result of the performance of the B_c tree method is somewhat worse than the result in [4] because we take into account the time difference between a random disk access and a sequential disk access.

Fig. 8 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. 7 except $B1 = B2$. θ is varied from 10 to 100. In general, a higher value of θ is not favorable to both the indexed methods. However, even when θ is 100, the SDB method is still far better than

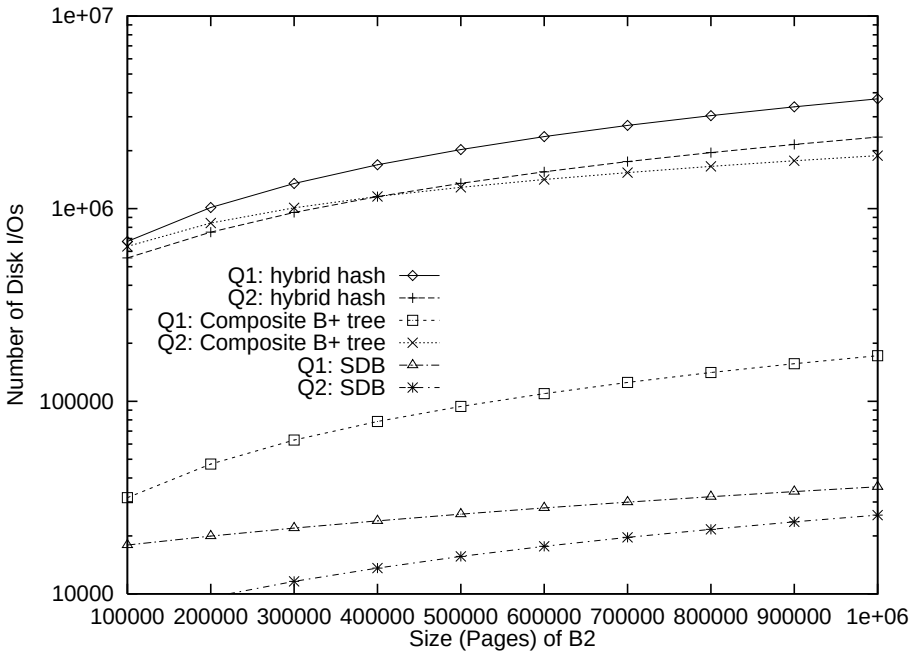

Figure 7: $\mid B1 \mid = 100k$ pages; $s = 5k$ pages; $\alpha = 0.00002$; $\theta = 20$.

the hybrid hash join.

These performance results show the performance advantage of the SDB method over a wide spectrum of conditions, as compared to the B_c tree method and the hybrid hash join. The performance advantage of the SDB method is especially evident when it is used in large relational databases in answering queries with small query results.

7 Summary Remarks

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

Queries are first executed in the summary relations. Bonding tables are used if the query contains joins. The query execution results from the summary relations are mapped to leaf nodes of addressing tables and are used to selectively load the pages of leaf nodes of the addressing table into the memory. Then, the join execution is completed by using the loaded portion of leaf nodes of the addressing table, similar to what has been described in

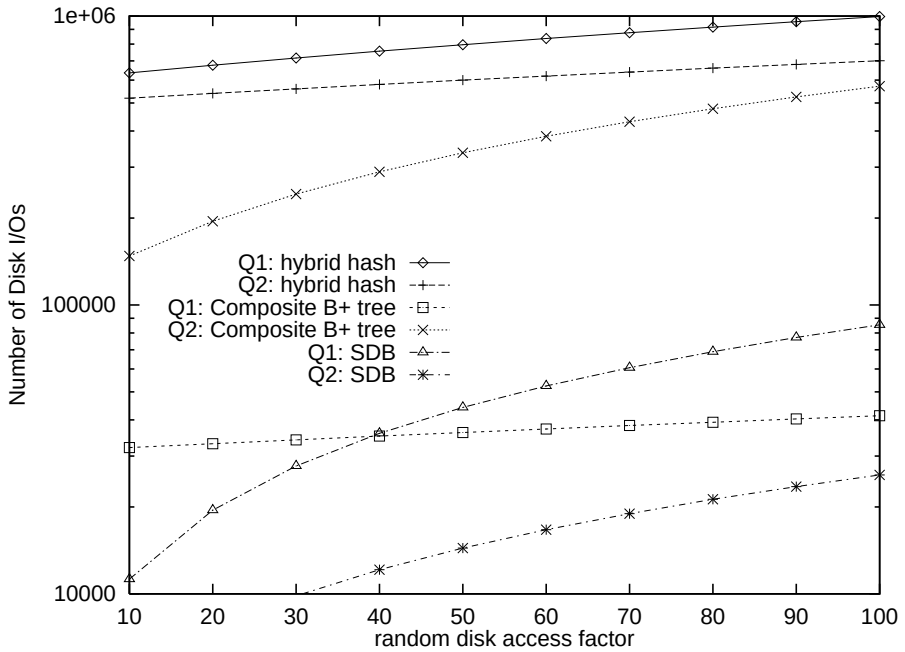

Figure 8: $\mid B1 \mid = \mid B2 \mid = 100k$ pages; $s = 5k$ pages; $\alpha = 0.00002$.

[4]. The summary relations and bonding tables are small compared to the size of all leaf nodes of the addressing tables (which has the same size as all B_c trees in the B_c method). Our approach uses a data structure that is slightly larger than that of the B_c tree method (whose data structure is much smaller than that in the join index method) but it provides much better join execution performance than the B_c tree method when the join selectivity is low and relations are large.

The SDB method is more than just a join method. A query with multiple joins and/or selections can be executed in the SDB. For a selection execution, the execution result of a query containing selections can be used to restrict the search scope of selection in the main database and thus, speeding up selection executions. In lieu of the large storage cost of setting up a B+ tree index for every attribute in the database, the SDB is a light-weight index for indexing all attributes in selection executions.

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" databases, such as the databases used over the World Wide Web and in mobile computing environments. These trends open the door for applying the SDB method. The excellent performance in answering queries with low selectivity joins

in large relational databases makes SDB method ideal for such applications.

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. There are also some research issues resulting from implementing the SDB method. For example, traditional query optimization strategy, in which selections are preferred before joins, fails to yield significant advantage when applied to the SDB method due to the small cost of executing a join (of course, only in the case when join selectivities are low and relations are large).

Appendix: Signature Method

A *signature* is a 0/1 bit pattern obtained by the hashing of data values. Fig. 9 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:

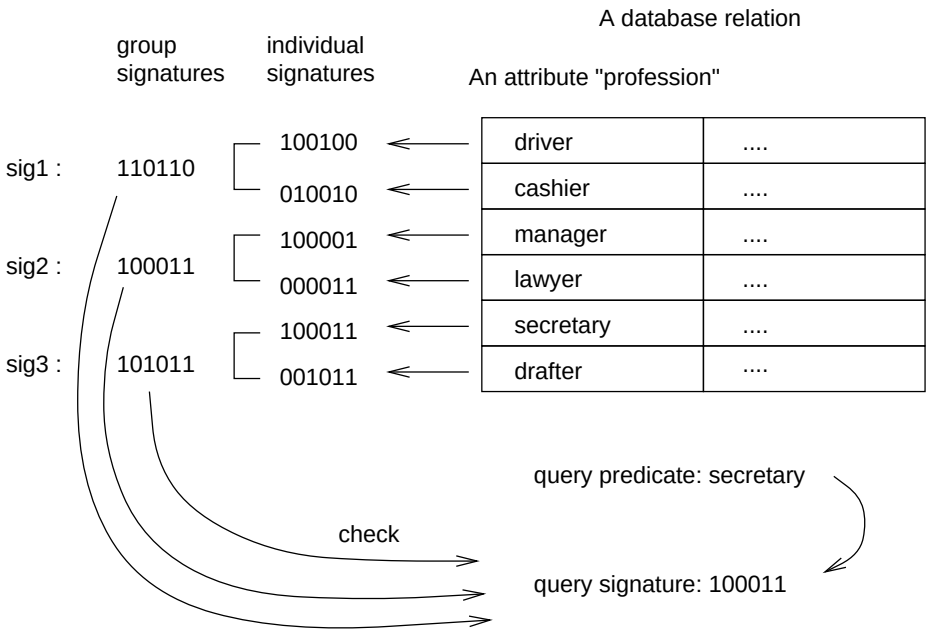

Figure 9: Individual signatures, group signatures and the query signature.

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

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 100, 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.

The signature is used to reduce the search space [5] and it is used solely for attributes with text data domain such as name, address, descriptions, etc. Numerical data such as weight and price, are generally not hashed into signatures because they are often used in range queries and hashed signatures usually cannot preserve the order and range property in the original numerical values. Signature method has been extensively investigated before, and we prefer using the signature method in *SDB* for its efficiency in representing text values.

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

[4] 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.*

[5] D. L. Lee, C.-W. Leng. "Partitioned Signature File: Design Issues and Performance Evaluation". *ACM Transactions on Information Systems, Vol.7, No.2, pp. 158-180, April 1989.*

[6] 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.*

[7] 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.*

[8] 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.*

[9] 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.*

[10] 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.*

[11] 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.*

[12] 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.*

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

[14] 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.*

[15] 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.*

[16] 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.*

[17] 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.*

[18] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: "An Efficient Data Clustering Method for Very Large Databases". *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data, pp. 103-114, June 1996.*