Fast Join Execution Using Summary Information in Large Databases

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

September 22, 1997

Abstract

It is well known that a query execution in relational databases is not fast and join execution is generally the most expensive operation in query executions. All three major join methods, namely, nested loops, sorting, and hashing have been perfected to an extent that any further improvement in these methods will enhance the performance of join execution only marginally. Even with the advent of parallel database, the enhancement in the performance of join execution is limited: join execution is often I/O bounded and an increase in the number of processors can only have a marginal effect after a certain point. The use of join index to speed up join execution is also limited due to its huge storage requirement.

In this paper, we propose to use summary information to expedite query execution, especially join execution, in large databases. Summary information is stored in a summary database which forms the upper layer of a multi-layer database. The size of the summary database is much smaller than the size of a join index. An interesting feature of our approach is that it allows the full set of query processing, including join operation, to be carried out in the summary database. The results of the query processing in the summary database are used to narrow down the search efforts in the main relational database and can greatly expedite the overall query execution in the main database.

The main contribution of this paper is that it proposes a mechanism to perform join using summary information for performance enhancement in query execution in relational databases. We present a performance analysis that shows that the proposed technique can greatly speed up query execution, especially join execution.

key words: relational database, large database, multiple layer, join, signature, grouping, performance.

1 Introduction

It is well known that the performance of query processing in large relational database is far from satisfactory. Join is one of the most expensive relational operations. Three major classes of join execution methods, nested loops join and its variation nested blocks join, sort-merge join and its many variations, hash join and its many variations, have been improved and perfected to such an extent that further improvements of these methods will only bring marginal effects on performance of join execution. Parallel join holds promises to further enhance performance of join execution. However, since join execution is often I/O bounded, without deploying multiple I/O channels and disks, simply increasing the number of parallelly executing processors will only have a marginal effect on the performance of join execution when the number of processors reach a certain level.

Join index was proposed [4, 16] to speed up join execution. It is essentially a pre-computed join to turn a join between two relations R and S into a selection in a join index. Join index is a two-attribute relation. Each tuple of a join index has two entries. One entry is a pointer to a tuple in R and the other entry is a pointer to a tuple in S. Use of join index can greatly enhance the performance of join execution. However, join index is a huge data structure. The huge size of join index limited the use of it.

This paper proposes a new *multi-layer database*, MDB, which can greatly enhance the performance of join execution in relational databases. An MDB is constructed from a relational database, called the *main database*, by adding a summary database to it. Each tuple in the summary database contains summary information from a group of tuples in the main database. The summary database is small because of the grouping of tuples and the small size of the summary database allows it to be searched very fast. The search in the summary database is not just a search as that is done in an index structure such as a B+ tree or a signature file. The main contribution of the paper is that it proposes to do search in the summary database as a full fledged relation query excution, including selection and join operations. This is made possible by adding bonding tables and join indexes to the summary database and by a strategy to group tuples in the main database.

The rest of the papers are presented as follows. Section 2 introduces the concept of the signature method and some terminologies which are necessary for further discussions. Section 3 describes the structures and purposes of major structural components of the MDB. Section 4 compares MDB approach to three related approaches, i.e., the join index, the signature file methods, and the multi-layer database proposed in [13]. Section 5

presents some optimization considerations for various components of MDB. Before Section 6, all the discussions are for text data and signature is used as a way to embody summary information. In Section 6, the treatment of numerical data in the summary database are discussed. Section 7 gives a performance comparison between a hybrid hash join in a traditional relational database and in a MDB. Finally, Section 8 presents the concluding remarks.

2 Background

2.1 Signature Method

Signature method has been extensively investigated since early 80's [2, 3, 5]. Here we give a brief description of the signature method. A *signature* is a 0/1 bit pattern obtained by hashing data values. Several signatures can be superimposed to form a group signature. Figure 1 shows a relation with an attribute "profession". Each value in "profession" is hashed into a signature, called an *individual signature*. Then, some individual signatures are grouped and superimposed to form *group signatures*, *sig1, sig2*, and *sig3*. Only group signatures are stored in ML. Let's consider the following query:

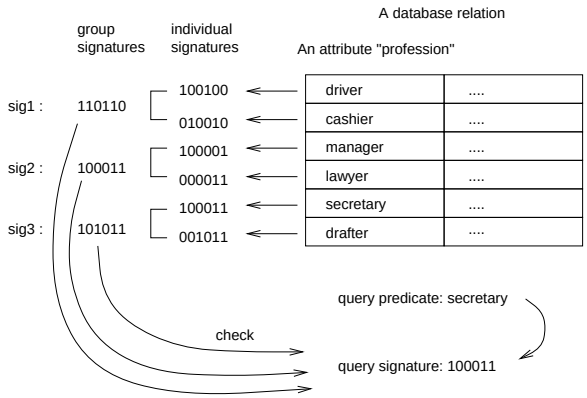


Figure 1: Individual signatures, group signatures and a query signature

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

The value "secretary" in the query predicate can be hashed into a signature *qsig*, called a *query signature*. *qsig* is checked against group signatures. A match is defined as each bit of 1 in *qsig* be matched by a bit of 1 in the same position of the group signature.

The group of values that generate *sig1* cannot possibly contain value "secretary" because *qsig* does not match *sig1*. Although *qsig* matches *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 size of a group is 100, and a lot of group signatures are not matched, then a significant amount of search can be reduced.

secretary	100011	⟵	*query signature*
*sig*1	110110	⟵	*not match*
*sig*2	100011	⟵	*false match*
*sig*3	101011	⟵	*true match*

Signatures are typically used for attributes with text data domain, such as names, descriptions, etc. Numerical data values such as weight, distance, etc., are not hashed to signatures because they are often used in predicates of range queries. For convenience of discussion, let's only deal with text data. Treatment of numerical data will be discussed in Section 6.

2.2 Terminologies

An MDB can be conceptually separated into two parts: the main database and the *multiple layered upper structure, ML*. At each layer of ML there is one small *summary database*. A summary database contains summary information from the main database. In this sense we call the main database a *base database* for the summary database. Summary information can be further extracted from layer 1 database to form a summary database in layer 2. A base database is a more general term than the main database because if MDB has three layers, a layer 1 database can be the base database for a layer 2 database.

A relation in the summary database, called a *summary relation*, contains summary information extracted from a relation in the base database. This relation in the base database is called the *base relation* for the summary relation.

In general, MDB can have more than two layers. At layer 0, there is only one database, i.e., the main database. Though not essential to our idea, at each layer of ML, the small size of the summary database allows it to be duplicated to introduce parallel executions of queries.

For convenience of discussion, let's assume that there is only one layer in ML, layer 1, and there is only one summary database in layer 1. In this case, the entire MDB has only two layers, layer0 and layer1; the main database is the only database in layer 0, and the term main database and base database can be used interchangeably.

3 Proposed Multi-Layer Database

A multi-layer database MDB consists of two parts: a main database and a summary database. A schematic diagram of an MDB is shown in Figure 2. Query executions are first carried out in the summary database. This is a full fledged relational query execution. In particular, join operation is allowed to be executed in the summary database. This gives approximate locations of the data of interests in the main database and will be done very fast since the the summary database is very small. After that, results are mapped to the main database and query executions are done in the main database. Query execution in the main database will be done very fast since the search in the summary database has already revealed the approximate and possible locations of all data of interests in the main database.

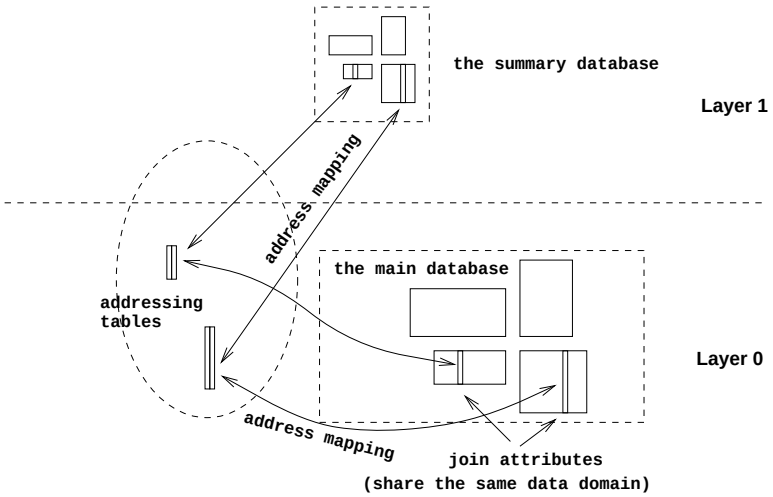


Figure 2: Architecture of an MDB

Three challenging problems need to be addressed. First, how to design the summary database so that it is small yet it can provide accurate information about locations of relevant data in the base database. Second, how to organize signatures in the summary database so that relational operations, especially join, can be performed efficiently on them. Third, once search is done in the summary database, how to map results to the main database to reduce search space in the main database. The first problem is solved by using group signatures, the second by organizing group signatures into a relational database like structure, and augmenting it with bonding tables and summary join indexes. The third problem is addressed by providing addressing tables. Next, we explain these concepts in details.

Summary Database: The summary database is comprised of summary relations. A summary relation is a complete database relation. This means that it has all the attributes that its base relation has. For each join attribute [1] in each relation of the main database, a summary relation is constructed that becomes a part of the summary database. A tuple in a summary relation corresponds to a group of tuples in its base relation.

The grouping criterion to group tuples in the base relation is based on the grouping of values in one of the join attributes of the base relation. To be more specific, first values in a join attribute in the base relation is partitioned. Tuples in the base relation are then grouped according to this partition. For each group of tuples and at each attribute, a group signature is obtained that becomes the value at this attribute in the corresponding tuple of the summary relation. This is shown in Figure 3.

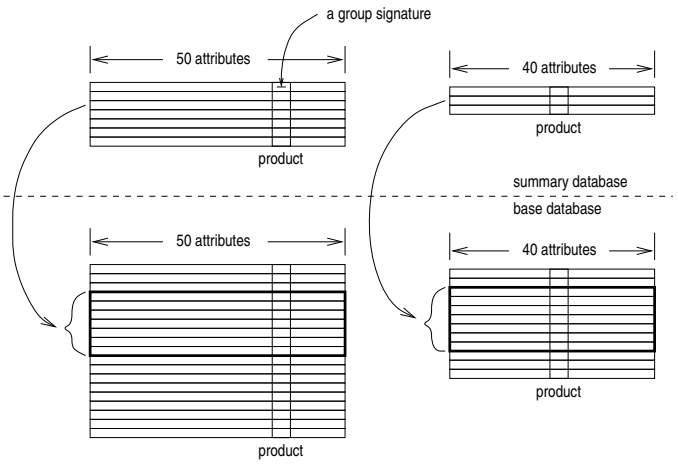


Figure 3: Correspondence between base relations and summary relations

Each text value in an entry at an attribute of a tuple in a group of tuples in the main database is hashed into an individual signature. All individual signatures from the same attribute in this group of tuples are superimposed to form a group signature. This group signature becomes the entry value at the same attribute of the corresponding tuple in the summary relation. A tuple in a summary relation is a tuple of group signatures.

A summary relation has only one base relation. Because a base relation may have one or more join attributes, a base relation can have one or more summary relations. Different summary relations of the same base relation are connected by bonding tables (discussed later). A join index is used to connect two different summary relations which are belong to two different base relations that are joinable. To avoid confusion, let's reserve the term

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

"join index" to be the one that is directly applied on the main database as proposed in [4], and let the join index that is applied on the summary database be called a *summary join index*. A bonding table is used to map results of selection or other join operations from one summary relation to another summary relation that share the same base relation. A summary join index is used to perform join between two different summary relations that stem from two different base relations that are joinable.

A summary database equipped with bonding tables and summary join indexes allows implementation of a full set of relational operations, including selection and join.

If on the average a tuple in the summary relation is generated from a group of 100 tuples from its base relation, and each entry value in its base relation takes approximately the same space as a signature. Then the size of a summary relation will be about 1% of its base relation. If on the average there are three join attributes in each base relation of the base database, then the total size of the summary database is about 3% of its base database. This is a very small storage overhead compared with the large size of a join index. The small size ensures that a query execution in the summary database can be done much faster than a search in a join index.

Consider a database which has two relations as shown in Figure 3, one relation *market* has 50 attributes and $100,000$ tuples, and the other relation *supplier* has 40 attributes and $200,000$ tuples. *product* is the join attribute in these two relations. The summary database contains summary relations $mark_S$ and sup_S, derived from *market* and *supplier*, respectively. The average size of groups in constructing $mark_S$ from *market* is 50, and is 100 for constructing sup_S from *supplier*. The number of tuples in $mark_S$ is $100,000/50 = 2,000$ and in sup_S is $200,000/100 = 2,000$. Let's consider the following query:

$$
\begin{array}{ll}
select & market.sale_price,\ supplier.supply_price \\
from & market, \\
 & supplier \\
where & market.product = supplier.product\ and \\
 & market.category = \text{“}homedevice\text{”}\ and \\
 & supplier.class = \text{“}electronic\text{”}.
\end{array}
$$

Let the size of all addresses, signatures and text values be 10 bytes. Each tuple in *market* is $50 \times 10 = 500$ bytes, and in *supplier* is $40 \times 10 = 400$ bytes. Joins are performed using the nested block method. When performing join the relation *market* or $mark_S$ is the outer relation and *supplier* or sup_S is the inner relation. Assume that the execution is done on a machine with available memory space of 10 MB. With assumed page size of $2,048$ bytes, 10MB of memory space is $5,000$ pages. The writing of join results is the same for both methods and is therefore ignored in comparison of disk I/Os.

The size of *market* is $(100,000 \times 50 \times 10)/2,048 = 24,414$ pages. This volume of data can be divided into 5 equal sized partitions. Each partition can reside entirely in memory. So relation *supplier* has to be read 5 times, and total number of disk reads to relation *supplier* is $(5 \times 200,000 \times 40 \times 10)/2,048 = 195,313$ pages. So the total number of disk reads to do a nested blocks join is $219,727$ pages.

When a summary database is used, the join is first performed between attribute *product* in summary relations $mark_S$ and sup_S. Disk reads to $mark_S$ are $(2,000 \times 50 \times 10)/2,048 = 489$ pages. Since 489 pages can reside entirely in memory, only one scan to sup_S is needed. Disk reads to sup_S are $(2,000 \times 40 \times 10)/2,048 = 391$ pages. If 60 tuples in $mark_S$ match wirh 40 tuples in sup_S, then there are $60 \times 50 = 3,000$ tuples in *market* and $40 \times 100 = 4,000$ tuples in *supplier* need to be read from the main database.

Let's assume that for each tuple, a page is read from the disk. Under this assumption, the number of disk reads to *market* is $3,000$ pages. Since $3,000$ pages can reside entirely in memory, the most conservative way to ensure sequential disk accesses is to simply scan *market* and *supplier* once. This practice results in $489 + 391 + 24,414 + 39,063 = 64,357$ pages of disk reads. This number is much smaller than the number obtained without using the summary database.

It should be noted that the MDB approach is independent of other methods like inverted indices and hash joins, and can be used in conjunction with all these methods to enhance the performance of join executions.

Bonding Table: Each of the summary relations that stems from the same base relation contains group signatures, and groupings are done based on values in different join attributes. Once the summary relations are constructed, there is no way to establish connections between these summary relations by directly examing these summary relations, except to go through the grouping process again in the base relation. Bonding tables are constructed to solve this problem.

A base relation may have more than one join attributes, so more than one summary relations may be constructed from the same base relation and a bonding table is needed between each pair of these summary relations.

Every row of the bonding table has two components each contains an address of the corresponding signature in one of the two summary relations, as shown in Figure 4. The horizontal line segments in the base relation represent groupings of tuples in the base relation. Tuple 2 in summary relation S_A corresponds to tuples b and c in summary relation S_B, and this correspondence is recorded in the bonding table.

Summary relations are much smaller than their base relations, and a base relation usually only has a few join attributes. Though it is conceivable that in the worst case, the number

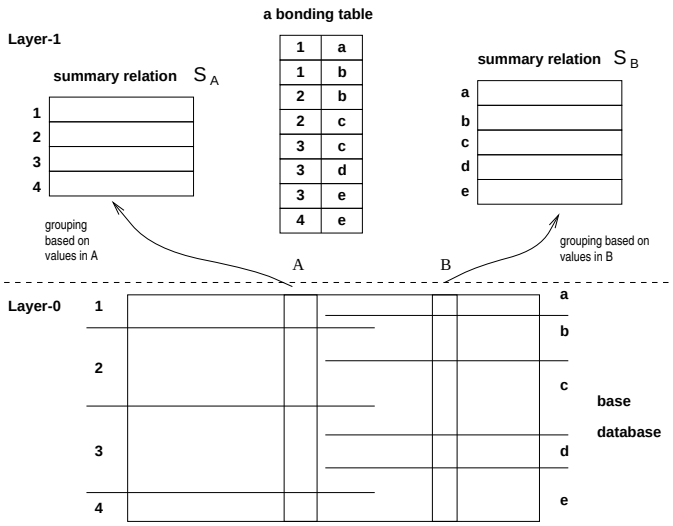


Figure 4: A bonding table between two summary relations

of rows in a bonding table could be the cross product of the numbers of tuples in the two summary relations that it connects, it will always be much smaller than the size of a join index [4]. In addition, the groupings that are used to build summary relations can be adjusted to reduce the size of a bonding table. In comparison, this flexibility does not exist in a join index. This will be discussed in more details in Section 5.

Addressing Table: After searching the summary database, a set of tuples are found that are relevant to the query. These tuples represent approximate and possible locations of the data of interests in the base database. However, if no special mechanism to map these tuples to the base relations, the entire base relations have to be scanned and the benefits of using the summary database are lost. An addressing table provides such a mechanism to map the tuples in a summary relation to tuples in the base relation.

Each entry in an addressing table has two components: an address of a tuple in a summary relation and a set of addresses of tuples in the base relation. The connection between addresses in these two components is such that the tuple in the summary relation contains group signatures at each attribute and that is obtained by superimposing individual signatures from the corresponding attribute in the group of tuples from the base relation.

After query processing is done in the summary database, addressing tables are used to map the resulting tuples in the summary database to the corresponding tuples in the base database.

Size of the addressing table could dominate the size of ML. If a base relation has $100,000$ tuples and three join attributes, then three addressing tables are needed each containing addresses of $100,000$ tuples. However, typically, the number of join attributes is much less

than the total number of attributes in a relation, so the total size of addressing tables will be far less than the total size of the base relation. Further reduction of the size of addressing tables will be discussed in Section 5.

4 Relation to Existing Works

A comparison among join index, materialized view, and hybrid-hash join by Blakeley et al. [6] found that when the data size is huge and the selectivity is low, the performance of the hybrid-hash join is the worst, the materialized view is better, and the join index is the best. However, a join index is a huge data structure. In the worst case, a join index can have $|| R || \times || S ||$ tuples, where $|| . ||$ represents the number of tuples in the relation. Because of its huge size, join index is not used in common relational databases with text and numerical data values, but used chiefly in spatial database [7, 10]. The huge size of the spatial data and the complexity of query execution in spatial database justify the use of a join index. We are not considering a materialized view here because it is even larger than a join index. Signature methods have been investigated extensively since early 80's [2, 3, 5] and is primarily used in text retrieval. Some have proposed to use signature file as an indexing structure to do preprocessing to reduce search space in the relational database [5]. However, they did not consider doing join operation in signature files.

Multiple layered database has been proposed before [11, 13, 14, 15]. However, the term "multiple layer" have been used differently and multiple layers have been used to improve modeling capability or database security. Only [13] proposed a multiple layered database structure to do preprocessing in the upper layer of the database to reduce search effort in the lower layer. However, a join operation is avoided in upper layer in [13].

The main differences between our approach and Han et al. [13] are that in our approach, there is a complete relational database in each layer (this is not mentioned in [13]), and join operations can be carried out in upper layer databases (this is explicitly avoided in [13]). In addition, though not essential to our idea, signatures are used to store summary information for text data to reduce size of upper layer databases.

5 Implementation Considerations

Grouping of values: Tuples in a base relation are grouped based on groupings of values in one of its join attributes. Let A be a join attribute with data domain D in relation R and S_D be the set of all values in all attributes of various base relations that share the same data domain D. The values in S_D are partitioned into groups and each group is given a unique identification number.

Then, from each group in S_D, values that belong to A in R are collected to form a group. This group of join attribute values determines a group of tuples in R. This process is repeated for each base relation that has a join attribute that share the same data domain as that of S_D. Each of the groups in each relevant base relation that comes from the same group in S_D bear the same identification number.

These identification numbers are entered into the same row of a summary join index. This approach establishes the connection between tuples in different summary relations that stem from different base relations that are joinable.

Groupings can be done with the help of concept hierarchies. A concept hierarchy [8, 9, 12, 13] is a tree-like structure. Each concept is represented as a node 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.

profession
driver
engineer
drafter
salesman
manager
lawyer

job
driver
secretary
lawyer
legal aid
drafter
manager
technician

$\Longrightarrow$

groups in profession	**groups in job**	**sets**
driver salesman	driver	out worker
engineer drafter	technician drafter	tech worker
manager	manager secretary	office worker
lawyer	lawyer legal aid	legal worker

In this example, attribute "profession" is a join attribute in R and "job" is a join attribute in S. All distinct values from these two join attributes are put together into a set. Let this set be partitioned into four groups: "out worker", "tech worker", "office worker", and "legal worker". Then, from each group, values that belong to "profession" are taken out to form groups in "profession". This process is repeated for attribute "job". Correspondence between groups of "profession" and groups of "job" is established through the same set of groups in S_D. For example, we can assign the same identification number to group {*manager*} in "profession" and group {*manager, secretary*} in "job" because they are all from the same group "office worker" in S_D.

This approach of grouping establishes one-to-one correspondence between tuples of different summary relations that stem from different base relations that are joinable, and minimizes the size of summary join index. It could happen that for one group of S_D, one summary relation has too many corresponding tuples and another summary relation has too few. In this case, a further partition of tuples of the first summary relation that corre-

sponding to this group is necessary, and the summary join index no longer holds one-to-one correspondence between tuples of the two summary relations. The partition of values in S_D should be done carefully to ensure each summary relation involved to have approximately equal number of tuples in each partition. The performance of MDB approach is sensitive to this partitioning.

Reducing the size of addressing tables: Conceptually, each entry in the addressing table should look like a tuple: $(address_{high}, address_{low})$, where $address_{high}$ is an address of a tuple in a summary relation and $address_{low}$ is an address of a tuple in the base relation. However, only one $address_{high}$ is needed for each group of $address_{low}$. Therefore, the bulk of an addressing table is comprised of addresses of the base relation and it is important to reduce the total size of base relation tuple addresses in an addressing table.

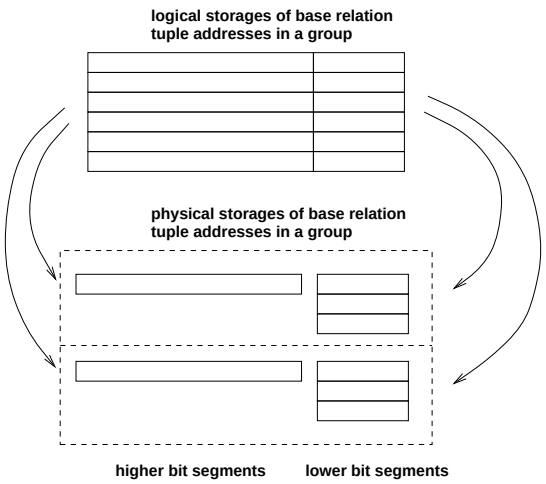


Figure 5: Bits segmentation in address to reduce the storage requirements

A base relation tuple address can be divided into two parts, for example, a higher bit segment and a lower bit segment, as shown in Figure 5. Suppose that the length of a base relation tuple address is 32 bits, the higher bit segment has 23 bits, and the lower bit segment has 9 bits. Let there be a total of 100 base relation tuples in a group, and there are 7 distinct higher bit segments and 100 distinct lower bit segments. These segments can be partitioned into 7 groups, each has a distinct higher bit segment and all its corresponding lower bit segments. So there are about 14 lower bit segments per group. The total space they occupy is $7+7\times14\times9 = 1,043$ bits. Comparatively, simply storing these base relation addresses requires $100\times32 = 3200$ bits. So using this approach reduces the size of addresses up to 67%. These numbers are given somewhat arbitrarily. However, they do illustrate a fact that clustering base relation tuples on disk can reduce the size of addressing tables.

After summary relations are constructed, tuples on disk can be physically re-arranged

according to grouping of the summary relation to reduce the size of addressing tables.

When all of the corresponding base relation tuples of a summary relation tuple are on a few physically closely located disk blocks, a way to further reduce the size of an addressing table is to store only the highest and the lowest addresses of these tuples. When these tuples are read from the disk, a simple filtering step is done to select tuples belong to this group.

In the worst case, the size of an addressing table is close to the size of one column in the base relational table. Since usually the number of attributes in a relation far exceeds the number of join attributes, the total size of addressing tables is still far less than the base relation, even in the worst case.

Updating MDB: Updates to summary relations are necessary when there are insertions, deletions, and change of values to entries of base relations. Updates in summary relations can be carried out immediately after the updates have happened in the base database. However, to reduce the amount of processing resulted from the updating of the summary database, the updates can be delayed and done in batches (or in period of low loads) depending upon the underlying application.

Specifically, if there are deletions in the base relation, no updates to the summary relations are necessary because deletions mean that some of the approximate and possible locations of data of interests discovered by query processing in summary relations no longer hold data of interests. This only means the effectiveness of using the summary database is reduced because that some false searches to the base database may be done. If there are insertions and changes of values in the base relation, to maintain consistency, updates to summary relation are necessary. However, for performance reasons, these changes may be propagated to the summary database in lazy manner because many real-life applications can work with static data.

6 Attributes with Numerical Data Domains

Treatment of numerical values in MDB is similar to that of the text values. Instead of group signatures, statistics like average, minimum, and maximum values are used to represent a group of numerical values in a compact way. Partitioning by range or clustering, such as BIRCH [17], can be used to group numerical values.

For rational numbers, the groups should be such that between any pairs of values within the same group lies no values belonging to other groups. This property is called the "convex" (For data values in 2 dimensional space, same sized and shaped rectangular boxes could be used to provide a convex grouping) in mathematics and is needed to process range queries such as all products with price below $1.2 per piece. Let us use an example to explain range

query processing in MDB. In the following query, values of attribute *product* are selected based on the price difference in their values in relation *market* and *supplier*:

$$\begin{array}{ll} select & market.product \\ from & market,\ supplier \\ where & \mid market.price - supplier.price \mid > 0.35 \end{array}$$

Let's Assume that all product prices in *market* and *supplier* are between \$0.5 and \$1.0. Price range for grouping is \$0.1 and there are five price groups between \$0.5 and \$1.0 as shown in Figure 6. During a query execution, if some prices in one group in *market* and some other prices in another group in *supplier* satisfy the query predicate, these two groups are considered matched. For example, *market.group_1* is matched with *supplier.group_4* and *supplier.group_5*, but not *supplier.group_3*, since none of the prices in *market.group_1* matches with prices in *supplier.group_3*.

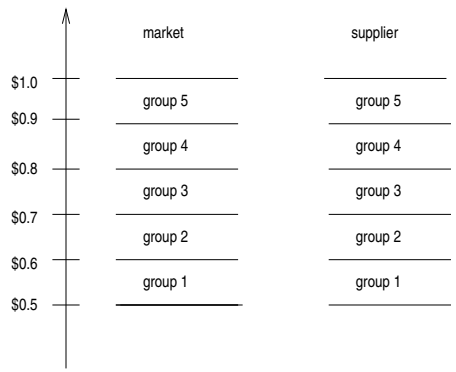


Figure 6: Groups of prices in *market* and *supplier* relations

7 Performance Analysis

We use the query given in Section 3 as a typical query to make comparisons, which are made between a relational database that has B+ indexes, to a MDB that has this relational database as its main database, and also has B+ indexes both on the summary database and on the main database. B+ indexes are used for selection operations before join and hashing is used in the partition phase of the join. For MDB, hash joins are carried out both at the summary database and at the base database. Comparisons are made for disk I/Os only and the unit of comparison is number of pages.

t		average number of tuples in each relation
g		group sizes for generating group signatures
a	50	average number of attributes in a relation
e	4	average number of bytes of a data entry
i	4	number of bytes in an address
s	4	number of bytes in a signature
p	2,048	number of bytes in a page
M	5,000	available memory pages
F_h	2	space overhead factor for hashing [6]
m	$t * a * e/p$	number of pages in a base relation
n	$(t * a * s)/(g * p)$	in MDB, number of pages in the summary relation
b	$(t * f_b * i)/(g * p)$	in MDB, number of pages in a bonding table
r	$(t * i * d)/p$	in MDB, number of pages in an addressing table
FO	$\sqrt{t/100}$	B+ tree fan out.
f_b	$\sqrt{g}$	in MDB, bonding table size factor
d	$1/\sqrt{\sqrt{g}}$	in MDB, size reduction ratio for addressing tables d
v	$t/10$	selection selectivity v in base database
v_s		in MDB, selection selectivity in summary relation v_s
v_j		in MDB, join selectivity v_j in summary relation
v_{ms}	$v_j * \sqrt{g}$	in MDB, selection selectivity in base relation

Table 1: Variables and values used in performance comparison

Assumptions Variables and their assumed values used in performance comparisons are listed in Table 1. Each base database relation is assumed to have a attributes, t tuples, and each tuple has $a * e$ bytes, so each base database relation has $t * a * e/p$ pages.

Formulas We adopt the formula in [6] for computing IO_{ii}, the number of pages accessed when accessing k_f tuples in a relation having m_f pages and n_f tuples via an inverted B+ index of two or three levels [6]:

$$IO_{ii}(k_f, m_f, n_f) = [Yao(k_f, m_f, n_f) + Yao(k_f, n_f/FO, n_f) + T3(k_f, n_f), \quad (1)$$

where

$$T3(k_f, n_f) = Yao(Yao(k_f, n_f/FO, n_f), n_f/(FO * FO), n_f/FO). \quad (2)$$

The formula $Yao(*)$ is the number of disk I/Os without any search overhead through indexs, and is given by Yao, S. B. [1]:

$$Yao(k_f, m_f, n_f) = m_f - m_f * \prod_{i=1}^{k_f} \frac{n_f - (n_f/m_f) - i + 1}{n_f - i + 1}. \tag{3}$$

Performance (B+ index, Hybrid hash join): The number of disk I/Os for two direct selection operations in two relations is:

$$C_{hs} = 2 * IO_{ii}(v, m, t); \tag{4}$$

Hashing in the hash join operation can be combined with the writing phase of the selection operation. After the selection operation, a hybrid-hash join operation needs to be performed on join attribute "product". Let

$$B_h = max(\ 0,\ \frac{F_h * v * a * e/p - M + 2}{M - 2}\ ), \tag{5}$$

Tuples in each operand relation are hashed into $B_h + 1$ buckets, each has at most $(M - 2) * p/(2 * a * e)$ tuples (note the unit of M is page). The number of disk writes for writing pages of all buckets (except anchor buckets) of both relations to disk is:

$$C_{hw} = 2 * (B_h + 1) * (M - 2). \tag{6}$$

The number of disk read in joining phase is:

$$C_{hj} = 2 * (B_h + 1) * (M - 2). \tag{7}$$

Performance (MDB, B+ index, hybrid-hash join): The number of disk reads for two selection operations (for attributes $market.category =$ "$homedevice$" and $supplier.class =$ "$electronic$") in the two summary relations is:

$$C_{ss} = 2 * IO_{ii}(v_s, n, \frac{t}{g}); \tag{8}$$

The number of disk reads for accessing two bonding tables (to project the selected results from attribute $market.category$ to $market.product$ and from attribute $supplier.class$ to $supplier.product$) is:

$$C_{sb} = 2 * IO_{ii}(v_s, b, \frac{t * f_b}{g}); \tag{9}$$

Let

$$B_s = max(\ 0,\ \frac{F_h * v_s * a * s/p - M + 2}{M - 2}\ ), \tag{10}$$

Then, the number of disk writes for writing out the selection results is:

$$C_{ssw} = 2 * (B_s + 1) * (M - 2). \tag{11}$$

The number of disk reads to read in pages to perform join operations in the summary relation is:

$$C_{ssj} = 2 * (B_s + 1) * (M - 2). \tag{12}$$

When all the computations in the summary database have been completed, the results need to be mapped to the main database for further processing. The number of disk reads for accessing two addressing tables is:

$$C_a = 2 * IO_{ii}(v_j * g, r, t); \tag{13}$$

Query execution has to be repeated again in the main database with reduced search space. The number of disk reads for two selection operations in two main relations is:

$$C_{ms} = 2 * IO_{ii}(v_j * g, m, t) \tag{14}$$

Let

$$B_m = max(\ 0,\ \frac{F_h * v_{ms} * a * s/p - M + 2}{M - 2}\ ), \tag{15}$$

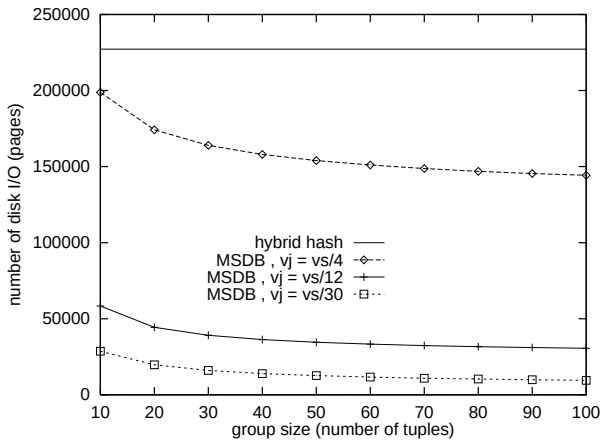
Then, the number of disk writes for writing out the selection results is:

$$C_{smw} = 2 * (B_m + 1) * (M - 2). \tag{16}$$

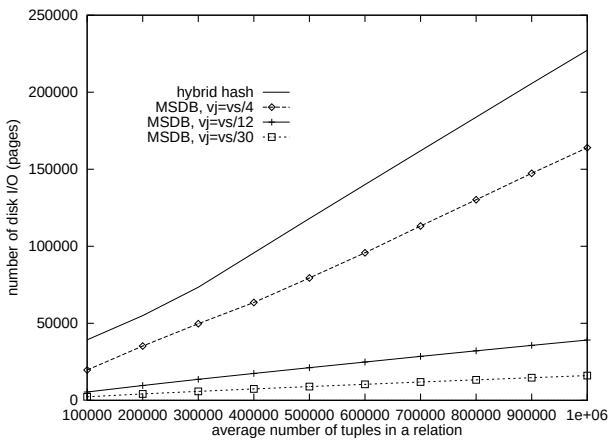
The number of disk reads to read in pages to perform join in the main relation is:

$$C_{smj} = 2 * (B_m + 1) * (M - 2). \tag{17}$$

Computation results: The selection selectivity and join selectivity are the most important factors in determining the join execution performance. Figure 7 shows comparisons when the selection and the join selectivities in the hybrid hash join stay the same, and in the MDB, join selectivities in the summary database are: $v_j = v_s/4, v_s/12, v_s/30$. and selection selectivities in the summary database are: $v_s = t/(\sqrt{g} * 4), t/(\sqrt{g} * 12), t/(\sqrt{g} * 30)$. 7(a) gives the results when $t = 1,000,000$, and group size g varies from 10 to 100. 7(b) gives results when the group size is fixed at 30 and the number of tuples t varies from 100,000 to 1,000,000.

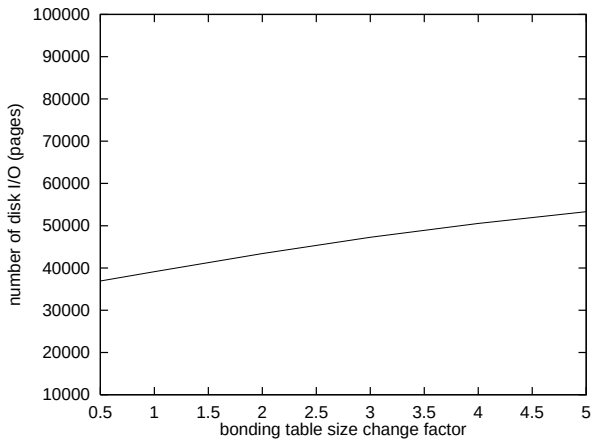


(a) Performances vs. group sizes

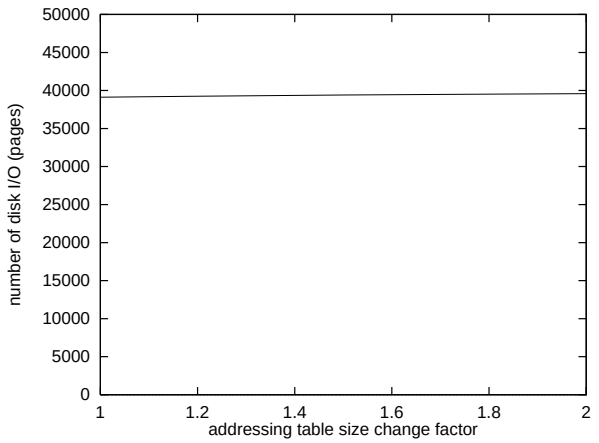


(b) Performances vs. # tuples

Figure 7: $v_j = v_s/4$, $v_j = v_s/12$, $v_j = v_s/30$



(a) bonding table



(b) addressing table

Figure 8: Effects of size changes of bonding tables and addressing tables

The join selectivities in the summary database in different computations in Figure 7 vary over a wide range of 56 times. Obviously, the lower the selectivity, the shorter the query execution time (measured in pages of disk accesses). The performance of MDB depends very much on the selection selectivity and the join selectivity in the summary database. However, even with the highest join selectivity ($v_j = v_s/4$, and $v_s = t/(\sqrt{g} * 4)$) in the summary database, the performance of MDB is still better than the performance of a hybrid hash join.

Another set of experiments is to vary the size of bonding tables and the size of addressing tables. The size of a bonding table can vary much more than the size of an addressing table. Figure 8(a) shows the effects of changing the size of bonding tables while the size of addressing tables stay the same as in Table 1. Figure 8(b) shows the effects of changing the size of addressing tables while the size of bonding tables stay the same as in Table 1. These results are obtained with relation size $t = 1,000,000$, group size $g = 30$, $v_j = v_s/12$, and $v_s = t/(\sqrt{g} * 12)$.

These experiments are carried out over a wide range of parameters and some trends can be clearly seen from this analysis. Figure 8 shows that sizes of bonding tables and addressing tables have a limited effect on the performance of the MDB. The group size affects the performance of the MDB. A large group size has a positive effect on the performance of the MDB. But the improvement of performance levels out as the group size approaches 100. Also, the join selectivity ratio between the curve with $v_j = v_s/4$ and the curve with $v_j = v_s/12$ is approximately the same as the join selectivity ratio between the curve with $v_j = v_s/12$ and the curve with $v_j = v_s/30$. However, the performance difference between the first two computations is obviously much bigger than the performance difference between the second two computations. This seems to suggest that in order for the MDB to have a good performance, the join selectivity in the summary database should be low, but the effect will gradually diminish as the join selectivity gets lower.

8 Concluding Remarks

In this paper a new database architecture, MDB, was presented and its major components and major implementation issues were discussed. In MDB, the size of the summary database is in general much smaller than the size of a join index. A query is first executed on the summary database whose results help greatly narrow down the search for relevant data in the main database for query processing in the main database, thus enhancing the performance of the overall query processing.

An interesting feature of the proposed mechanism is that it performs a complete query processing, including join execution, in the summary database. The relatively small size of

the summary database not only makes MDB approach more practical to implement, but also makes query processing fast. The small size of the summary database results from the grouping of tuples, but grouping of tuples complicates the implementation of the relational operations. We developed appropriate data structures to enable relation operations in the summary database.

We conducted an analytic study of the proposed method. The performance of MDB is especially good when the selectivity of the query is low, i.e., query that only returns small amount of information. This is similar to that of the joining index [6]. Overall, MDB approach is suitable for query-intensive large relational databases with low query selectivities.

As for further research, the grouping of data values, both for text values as well as for numerical values, is critical to the performance of MDB and more study is needed in this direction.

References

[1] S. B. Yao. Approximating block accesses in database organizations. *Commun. ACM 20, 4(Apr. 1977)*, pp. 260-261.

[2] R. Sacks-Davis, K. Ramamohanarao. A two level superimposed coding scheme for a partial match retrieval. *Information Systems, Vol. 8, No. 4, 1983*, pp. 273-280.

[3] C. Faloutsos. Signature files: Design and performance comparison of some signature extraction methods. *Proc. of the Int'l SIGMOD Conf. on the Management of Data, 1985*, pp. 63-82.

[4] P. ValDuriez. Join Indices. *ACM Transactions on Database Systems, Vol 12, No.2, 1987*, pp. 218-246.

[5] R. Sacks-Davis, A. Kent, K. Ramamohanarao. Multikey Access Methods Based on Superimposed Coding Techniques. *ACM Transactions on Database Systems, Vol. 12, No.4, 1987*, pp. 655-696.

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

[7] D. Rotem. Spatial Join Indices. *IEEE Proc. of 7th Int'l Conf. on Data Engineering, 1991*, pp. 500-509.

[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, July 1991, pp.77-94.

[9] J. Han, Y. Cai, and N. Cercone. Knowledge Discovery in Databases: An Attribute-Oriented Approach. *Proc. of 18th VLDB Conf., 1992*, pp. 547-559.

[10] Wei Lu, J. Han. Distance-Associated Join Indices for spatial range search. *IEEE Proc. of 8th Int'l Conf. on Data Engineering, 1992*, pp. 284-292.

[11] K. Smith, M. Winslett. Entity Modeling in the MLS Relational Model. *Proc. of the 18th VLDB Conf., 1992*, pp. 199-210.

[12] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. on Knowledge and Data Engineering, 1993*, 5:29-40.

[13] 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, May 1994. pp. 47-58.

[14] 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.

[15] F. Chen, R. S. Sandhu. The Multilevel Relational (MLR) Data Model. *Technical Report, ISSE-TR-95-101, 1995, George Mason University.*

[16] Z. Li, K. A. Ross. Fast Joins Using Join Indices. *Technical Report, CUCS-032-96, 1996, Columbia University.*

[17] 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,* Montreal, Canada, 1996.