A Physical Storage Model for Efficient Statistical Query Processing*†

WEE K. NG CHINYA V. RAVISHANKAR

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
Email: {wkn,ravi}@eecs.umich.edu

Abstract

A common approach to improving the performance of statistical query processing is to use precomputed results. Another lower-level approach would be to redesign the storage structure for statistical databases. This avenue is relatively unexplored. The objective of this paper is to present a physical storage structure for statistical databases, whose design is motivated by the characteristics of statistical queries. We show that our proposal enhances multi-attribute clustering efficiency, and improves the performance of statistical and aggregational queries. This customized structure reduces the amount of I/O incurred during statistical query processing, thus decreasing the response time.

1 Introduction

There has been much work to date on statistical database and statistical query processing. Such work covers the areas of data modeling (see survey in [16, 17]), query languages (see survey in [20]), extension of the relation model [8], the use of pre-computed results (summary data) [1, 3, 4, 7, 10, 14] and access methods [19]. Shoshani [18] and Michalewicz [12] have given a comprehensive introduction to issues in statistical and scientific databases.

In this paper, we are interested in the performance aspects of statistical queries. It is difficult to get good query processing performance in statistical databases for several reasons. Firstly, statistical databases exhibit high volume and high retention. Data volume increases monotonically because historical data are usually archived rather than discarded. Secondly, statistical queries are predominantly aggregational and thus data-intensive. As I/O latency is high, these factors are compounded in statistical queries by the I/O bottleneck problem.

A commonly adopted approach to alleviating the problem is through the use of precomputed results. In this approach, *elementary summary statistics* [3] such as partial sums, partial sum of products, etc. are precomputed for a set of attributes. The on-line computation of these results is time-consuming because a lot of I/O is involved. When summary statistics are used to satisfy such queries, a lot of effort is saved. Some issues in this approach include (1) finding a logical data model for these statistics, (2) developing a query language to access them, (3) the problem of statistics derivability, and (4) the physical organization and management of summary statistics.

Our approach does not involve summary statistics. Instead, we work at the physical organization level to improve the performance of statistical queries, and propose a new physical storage structure for statistical relations. This structure is motivated by the characteristics of statistical queries, and we show that it helps to reduce the amount of I/O incurred during the execution of statistical queries.

This paper is organized as follows: In the next section, we examine the idiosyncrasies of statistical data and queries. They are used to derive the design of the storage structure, described in Section 3. In Section 4, we demonstrate how to support standard database operations when this new storage structure is used. This is further elaborated in Section 5 on query processing. We show that little changes are required. In Section 6, we evaluate the performance of this structure with respect to query processing. Section 7 discusses previous works in the same area. Finally, we conclude the paper with some remarks on future work.

*This work was supported in part by the Consortium for International Earth Science Information Networking.

†The material contained in this paper may be covered by a pending patent application.

0-8186-6610-2/94 $04.00

2 Characteristics of Statistical Queries

Statistical databases differ from ordinary databases in several ways. The attributes of a relation in a statistical database may be grouped into two classes: *category* and *summary* [3, 4, 7, 8]. Category attributes are generally descriptive (non-numeric) and have discrete values that are known in advance. They are used in queries as access keys for retrieving tuples, and are rarely modified. On the other hand, summary attributes are usually numeric because they are usually the observed or measured values in some experiment or survey. They are used in the computation of statistics for statistical queries, and have a higher probability of being modified.

This classification of attribute domains is important as it reflects the differences in functional usage within statistical queries. Logical operators are usually applied to the category attributes, while statistical operations are performed on the summary attributes. As we shall see in the next section, our new storage structure for statistical relation captures and exploits this fact.

Since statistical queries are predominantly aggregational, their tuple access pattern is different from that of ordinary queries. A statistical query usually needs to access a group of tuples satisfying certain criteria. Aggregational queries in statistical databases often require access to *all* tuples having specified values in a subset of their attribute fields. Such queries can result in a proliferation of disk seeks since conventional database structures are not designed for clustering multiple attributes.

In a conventional database, tuples of a relation are physically clustered via a unique key called the *ordering key*. An access mechanism, such as a B^+ tree, is used to provide random access. The tuples are non-clustering with respect to the majority of category attributes. If tuples satisfying some search criteria are physically clustered together, fewer disk blocks will be accessed as multiple candidate tuples are found in a block, thus improving the response time of the query. We show in the following sections how our approach enhances multi-attribute clustering.

3 Proposed Storage Structure

We now turn to the issue of how to redesign the physical storage structure of a relation so as to accomodate the characteristic requirements mentioned in the previous section.

3.1 Attribute domain mapping

A statistical relation scheme $\mathcal{R} = C_1 \times \cdots \times C_n \times S_1 \times \cdots \times S_m$ is the *tuple space* containing the set of all possible tuples where C_i's and S_j's are category and summary attribute domains respectively. We will write $\mathcal{R} = \langle\langle \mathsf{C}, \mathsf{S} \rangle\rangle$. A statistical relation instance (or simply a relation) R, is a subset of tuples from the tuple space, i.e., $R \subseteq \mathcal{R}$.

Example 3.1 Table 3.1(a) depicts a statistical relation R which will be used to illustrate all the concepts discussed in this paper. There are four category domains C_1, C_2, C_3, C_4 and two summary domains S_1, S_2. ∎

The first step in defining the new storage structure is *attribute domain mapping*. We map each attribute value of a non-numeric category domain to a numeric value. The motivation for domain mapping will be explained in Section 3.3. Table 3.1(b) is the numerically mapped version of Table 3.1(a). The mapping of category domains is easy as they are usually of finite size with all possible attribute values known in advance. For every attribute $c \in C_i$, we map c into its *ordinal* position within the domain. For example, C_1={production, marketing, personnel} is mapped into $C_1 = \{0, 1, 2\}$.

3.2 Multi-attribute clustering

Consider Table 3.1(c), which is identical to Table 3.1(b) in content, except that the tuples have been reordered (and domain C_4 reordered). Notice that the attribute values under column C_1 form runs of 0's, 1's and 2's. Column C_2 exhibits similar behaviour except that the runs are shorter.

If the tuples are physically clustered as in Table 3.1(c), the probability of finding tuples whose ith attributes are identical being stored within the same block is very much higher than when they are clustered via a primary key in conventional style. Searching for tuples that have a certain category attribute value requires fewer block accesses because multiple candidate tuples are clustered in the same blocks. This reduces I/O and improves performance.

Compare this with the conventional approach via secondary indices. Because the tuples are clustered physically in a different order, the indices must point all over to locate the tuples. Thus, when retrieving a tuple via a secondary index, many random blocks are accessed, and that the same block be accessed more than once. As the latency of disk block I/O is high,

C_4	C_1	C_2	C_3	S_1	S_2
1	*personnel*	*worker*	*Spanish*	25	30
2	*personnel*	*manager*	*German*	28	20
3	*marketing*	*supervisor*	*Spanish*	24	35
4	*production*	*worker*	*Spanish*	32	25
5	*marketing*	*manager*	*German*	47	20
6	*marketing*	*worker*	*English*	38	25
7	*personnel*	*leader*	*German*	23	20
8	*personnel*	*supervisor*	*English*	37	40
9	*marketing*	*leader*	*German*	44	25
10	*production*	*supervisor*	*Spanish*	23	30
11	*production*	*worker*	*German*	32	20
12	*personnel*	*leader*	*French*	32	20
13	*marketing*	*worker*	*French*	34	40
14	*personnel*	*worker*	*French*	31	30
15	*marketing*	*leader*	*English*	28	35
16	*personnel*	*manager*	*French*	29	25
17	*production*	*manager*	*Spanish*	46	20
18	*production*	*supervisor*	*German*	25	20
19	*marketing*	*supervisor*	*English*	35	40
20	*personnel*	*leader*	*English*	26	30
21	*marketing*	*leader*	*Spanish*	27	30
22	*production*	*supervisor*	*English*	36	30
23	*personnel*	*worker*	*German*	29	35
24	*personnel*	*supervisor*	*Spanish*	38	30
25	*personnel*	*supervisor*	*French*	42	25
26	*marketing*	*worker*	*Spanish*	24	25
27	*marketing*	*manager*	*French*	27	30
28	*production*	*leader*	*Spanish*	22	30
29	*production*	*leader*	*German*	28	20

(a)

C_4	C_1	C_2	C_3	S_1	S_2
1	2	3	1	25	30
2	2	0	3	28	20
3	1	1	1	24	35
4	0	3	1	32	25
5	1	0	3	47	20
6	1	3	0	38	25
7	2	2	3	23	20
8	2	1	0	37	40
9	1	2	3	44	25
10	0	1	1	23	30
11	0	3	3	32	20
12	2	2	2	32	20
13	1	3	2	34	40
14	2	3	2	31	30
15	1	2	0	28	35
16	2	0	2	29	25
17	0	0	1	46	20
18	0	1	3	25	20
19	1	1	0	35	40
20	2	2	0	26	30
21	1	2	1	27	30
22	0	1	0	36	30
23	2	3	3	29	35
24	2	1	1	38	30
25	2	1	2	42	25
26	1	3	1	24	25
27	1	0	2	27	30
28	0	2	1	22	30
29	0	2	3	28	20

(b)

C_1	C_2	C_3	C_4	S_1	S_2
0	0	1	17	46	20
0	1	0	22	36	30
0	1	1	10	23	30
0	1	3	18	25	20
0	2	1	28	22	30
0	2	3	29	28	20
0	3	1	4	32	25
0	3	3	11	32	20
1	0	2	27	27	30
1	0	3	5	47	20
1	1	0	19	35	40
1	1	1	3	24	35
1	2	0	15	28	35
1	2	1	21	27	30
1	2	3	9	44	25
1	3	0	6	38	25
1	3	1	26	24	25
1	3	2	13	34	40
2	0	2	16	29	25
2	0	3	2	28	20
2	1	0	8	37	40
2	1	1	24	38	30
2	1	2	25	42	25
2	2	0	20	26	30
2	2	2	12	32	20
2	2	3	7	23	20
2	3	1	1	25	30
2	3	2	14	31	30
2	3	3	23	29	35

(c)

C_1	C_2	C_3	C_4	S_1	S_2
0	0	3	5	36	30
0	0	0	88	23	30
0	0	2	8	25	20
0	0	2	10	22	30
0	0	2	1	28	20
0	1	0	11	32	25
0	0	1	75	32	20
0	0	2	7	27	30
0	0	2	16	47	20
0	0	1	78	35	40
0	0	0	84	24	35
0	0	3	12	28	35
0	0	1	6	27	30
0	0	0	97	44	25
0	0	1	88	38	25
0	0	1	20	24	25
0	0	0	87	34	40
0	1	0	3	29	25
0	0	0	86	28	20
0	0	0	6	37	40
0	0	1	16	38	30
0	0	1	1	42	25
0	0	1	75	26	30
0	0	2	12	32	20
0	0	0	95	23	20
0	0	1	94	25	30
0	0	1	13	31	30
0	0	1	9	29	35

(d)

Table 3.1: A relation R and its transformations. There are four category domains in R: C_4,C_1,C_2,C_3, denoting the *employee number*, *department*, *job title*, and the *language spoken* respectively, and two summary domains S_1 and S_2, denoting age and income. Table (a) is the original relation. Table (b) shows the relation after every category domain has been mapped to integers. The sizes of the numerical domains C_4, C_1, C_2, C_3, are $100, 3, 4, 4$ respectively. Table (c) shows the relation ordered via the mixed-radix integeral order. Table (d) is the *compressed* relation of Table (c).

this method of access is expensive. Therefore, multi-attribute clustering alleviates this difficulty. It is to be noted that multi-attribute clustering applies to category domains only. The summary domains are not affected.

3.3 Tuple ordering scheme

What is the order that arranges the tuples in Table 3.1(c)? This is the lexicographical order. We also call it the *mixed-radix integral* order because we treat each tuple as a mixed-radix integer, and use the value of this integer as the primary and ordering key, as explained below.

The motivation for domain mapping should now be clear. With all attributes mapped to integers, a relation becomes a set of mixed-radix integers. These integers may then be sorted numerically into the form shown in Table 3.1(c).

Example 3.2 Continuing with our previous example, we note that the sizes of C_1, C_2, C_3, C_4 are $3, 4, 4, 100$ respectively. By adopting the sizes as the radices, the category portion of the last tuple $\langle 2, 3, 3, 23\rangle$ becomes $2 \times (4 \times 4 \times 100) + 3 \times (4 \times 100) + 3 \times (100) + 23 = 3200 + 1200 + 300 + 23 = 4723$. Thus, tuples may be compared and sorted by comparing their numerical values. We would like to repeat that the mixed-radix integral concept is applied to the category half of a tuple only. The summary portion tags along unchanged. ∎

3.4 Tuple compression

While lexicographical ordering enhances multi-attribute clustering, a characteristic requirement of statistical queries, its greatest side-benefit is that it also provides compression. Each of the runs of identical attribute values exhibits *value redundancy* that can be eliminated via simple compression techniques such as run-length coding [9]. However, a straightforward application of run-length coding on the vertical runs distorts the structure of a relation: We wish to retain the *table of tuples* definition of a relation. Hence, additional refinements are needed.

Example 3.3 Table 3.1(d) is the *compressed* version of Table 3.1(c). Notice the *horizontal* rows of leading zeroes in the tuples. By performing a *mixed-radix subtraction* between pairwise consecutive tuples of Table 3.1(c), one transforms the original vertical runs of identical attribute values into horizontal rows of leading zeroes. For instance, category portion of the first tuple in Table 3.1(d), $\langle 0,0,3,5\rangle$, is obtained by the subtraction: $\langle 0,1,0,22\rangle - \langle 0,0,1,17\rangle$, which are category portions of the second and first tuple in Table 3.1(c). The redundancies are retained, but the leading zeroes may now be encoded via run-length coding without sacrificing the tuple-structure of a relation. Thus, compression is achieved. ∎

Table 3.1(d) is the final storage structure for R. This structure not only exhibits multi-attribute clustering, but its storage requirements are reduced via compression. We shall refer to this new storage structure as the *Tuple Differential Structure*, or TD structure for short.

4 Standard DB Operations

How are standard database operations supported in the TD structure? In this section, we shall look at tuple access, insertion, deletion and modification. The next section discusses the support for query processing.

4.1 Access method

As tuples are now clustered under lexicographical order, a primary index for the relation uses an entire tuple as the search key. Figure 4.1 shows an order-3 primary B^+ tree index constructed for relation R. Notice the placement of tuples into disk blocks, which reflects the demarcations shown in Table 3.1(d).

There are two distinguishable parts of a tuple: the category portion and the summary portion, which we shall refer to as the *category sub-tuple* and *summary sub-tuple* respectively. Each block begins with a *head* tuple which is the numerically smallest category sub-tuple in the block. All tuples following the head tuple are difference tuples. Notice that the leading zeroes of the category sub-tuples are replaced by a number indicating the counts of the number of leading zero (run-length coding). Thus, the first difference tuple $\langle 2 \mid 3,5 \mid 36,30\rangle$ in block 1 is decoded into $\langle 0,0,3,5 \mid 36,30\rangle$. The head tuple can be added (via mixed-radix addition) to the differences to derive the actual tuples. For instance, block 2 begins with head tuple $\langle 0,2,1,28 \mid 22,30\rangle$ because the first difference tuple $\langle 0,0,2,1 \mid 28,20\rangle = \langle 0,3,1,4 \mid 32,25\rangle - \langle 0,2,3,29 \mid 28,30\rangle$. The purpose of starting a block with a head tuple is to restrict the scope of decompression to within a data block. If only a block is searched, the difference tuples may be decoded immediately without necessitating the decompression of all preceding blocks.

The primary index is useful only when the category portion of a tuple is completely available as the search key. When only some category attributes are known, secondary indices are needed. A secondary index requires a level of indirection between the attribute values and the data blocks where they might be found. For instance, the following provides the indirection for domain C_2: $(0 \mid 2,3,5),(1 \mid 1,3,5,6),(2 \mid 1,2,3,4,6,7),(3 \mid 2,4,5,7)$, which says that tuples whose $C_2 = 0$ are located in block $2,3,5$ (see the block demarcations in Table 3.1(c)). As the example relation R is too small, we are unable to construct a full-scale secondary index for any of the category attributes. Nevertheless, we may conclude that with the help of the primary and secondary indices, tuple access carries on as usual, even when the tuples are stored compressed under the TD structure.

4.2 Tuple insertion and deletion

How are tuple insertion and deletion supported in the database? Suppose we wish to insert tuple $t = \langle 1,0,3,6 \mid 20,20\rangle$, which differs from $\langle 1,0,3,5 \mid 47,20\rangle$ in the last attribute value. The primary index provides the means to locate the block which contains tuples that are physically ordered in the neighourhood of t. With this index, data block 3 is found to be the candidate block for inserting t.

block 3
⟨1,0,2,27 | 27,30⟩
⟨2 | 2,16 | 47,20⟩
⟨2 | 1,78 | 35,40⟩
⟨3 | 84 | 24,35⟩
⟨2 | 3,12 | 28,35⟩

⟼

block 3
⟨1,0,2,27 | 27,30⟩
⟨2 | 2,16 | 47,20⟩
⟨3 | 1 | 20,20⟩
⟨2 | 1,77 | 35,40⟩
⟨3 | 84 | 24,35⟩
⟨2 | 3,12 | 28,35⟩

The above reflects the changes to data block 3. Notice that only those difference tuples succeeding t are recomputed, and that the changes are confined to within the affected block. For tuple deletion, the primary index is similarly used to locate the data block and changes made within the block.

4.3 Tuple modification

In conventional database, tuple modification is performed *in situ*, i.e., right where the tuple is located due

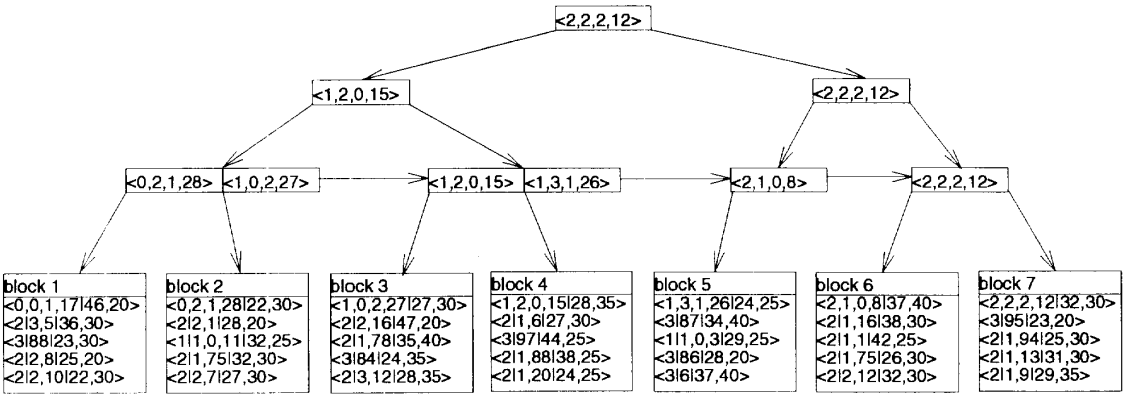

Figure 4.1: Primary index. The data blocks contain difference tuples in lexicographical order. Hence the search key is an entire tuple. Each block begins with a *head* tuple. All tuples following the head tuple are difference tuples where the leading zeroes are replaced by numbers (separated by a bar) indicating the counts. The summary portion of a tuple is also separated from the category portion by a bar.

to the tuple-wise storage structure of a relation. When tuples are stored in the TD structure, tuple modification can expected to be different. Suppose we wish to modify the first attribute of $\langle 1,0,3,5 \mid 47,20\rangle$ to get $\langle 2,0,3,5 \mid 47,20\rangle$. Due to the lexicographical ordering of tuples, the modified tuple would be physically far away from the pre-modified tuple in a different block. A tuple modification entails a deletion followed by an insertion, compared to the *in situ* modification in conventional database.

A closer look, however, reveals that the modification above has been made to a category attribute, which is rare in statistical databases (see Section 2). Statistical queries are by nature access-only; new statistical data are generated by reading a relation. Modifications, if any, are generally made only to summary attributes. Since summary attributes in the TD structure are stored tuple-wise (see Table 3.1(d)), tuple modification is still performed *right-where-it-is*.

In summary, standard database operations are the same even when the relation is stored in the new storage structure. The only difference being that the search key of the primary index is the entire tuple. All other indices are non-clustering and secondary, as in standard databases.

5 Statistical Query Processing

The primary objective of the TD structure is to reduce the amount of I/O for statistical queries, which generally involve large data transfers between main memory and secondary storage. In the next subsections, we examine some of the commonly encountered statistical queries.

5.1 Range query

We first look at a simple SQL query:

```
SELECT   employee.number, income
FROM     employee
WHERE    department = marketing OR
         department = personnel
```

which is translated into the following relational algebraic expression with respect to relation R shown in Table 3.1(a): $\sigma_{1\leq C_1\leq 2}(R)$, where C_1={production, marketing, personnel} has been mapped numerically to $C_1 = \{0,1,2\}$.

Strictly speaking, the above query is not the most appropriate illustration of a range query. However, it demonstrates the fact that range queries are usually translated into *exact-match* queries where the specified attribute assumes values within a consecutive range. In the example, we are looking for tuples whose C_1 attribute values are marketing or personnel, i.e., $C_1 \in \{1,2\}$. In any case, satisfying such queries requires accessing a large portion of the relation involved. With a secondary index constructed from C_1, we are able to locate tuples whose department attribute is marketing or personnel. The example illustrates the fact that when locating tuples via *any* attributes, we find that multiple candidate tuples matching the query selection criteria are physically clustered

in the same blocks because of the multi-attribute clustering feature of the TD structure. As a result, the actual amount of data blocks accessed is reduced.

5.2 Cross tabulation and aggregate query

Statistical queries are characterized by aggregate computations. One is usually looking for aggregates of some attributes of a group of tuples satisfying certain criteria. Suppose a query desires the average income of employees categorized by their department and job-title:

```
SELECT    job-title, AVG(income)
FROM      employee
WHERE     department = marketing AND
          (job-title = leader OR job-title = worker)
GROUP BY  job-title
```

In order to satisfy the above query, we need to (1) select tuples satisfying the multi-attribute selection criteria, (2) group tuples by job-title, and (3) compute the average income within each group.

The first step may be simplified into a combination of single-attribute retrievals. Using the secondary index for each attribute, the set of candidate tuples are retrieved. Retrieving tuples by a search key has been discussed in the previous section. In any case, a large portion of relation is accessed. Since the overall number of blocks of a relation stored under the TD structure is reduced, processing this query will be substantially faster than a conventionally stored relation.

In summary, reducing the amount of I/O increases the performance of queries involving large data transfers. The new storage structure realizes the reduction through multi-attribute clustering of tuples and tuple compression.

6 Performance Measurements

How good is the new storage structure in terms of improving the performance of statistical queries? Specifically, we want to know the following: (1) What reduction in disk block access is achieved on average per query? (2) What is the average reduction in the number of disk blocks accessed when locating tuples satisfying a selection criteria?

Test number	*1,5*	*2,6*	*3,7*	*4,8*
Data skew	*Yes*	*Yes*	*No*	*No*
Domain variance	*Small*	*Large*	*Small*	*Large*

Table 6.2: Test parameters. The two parameters, *data skew* and *domain variance*, give a total of four combinations for three sets of four tests. Tests $1, 2, 3, 4$ measure the number of disk blocks accessed on average per query for different relation sizes. Tests $5, 6, 7, 8$ measure the number of blocks required for database storage for different relation sizes.

6.1 Multi-attribute clustering

The first question concerns the multi-attribute clustering efficiency of the TD storage structure. This question is easy to answer: We have only to compare the average amount of I/O required for a typical query for a relation that is stored conventionally and in the storage structure. This raises two issues: What constitutes a typical query and a typical relation?

It is difficult to generate a typical "query" as there are many possibilities. To simplify things, we consider selects of the form $\sigma_{C_1=c_1, C_2=c_2, \ldots, C_k=c_k}(R)$, i.e., selecting a set of tuples whose category attributes satisfy certain values. Here, $k < n$ where n is the total number of category attributes and c_i's are randomly generated attribute values of C_i, $1 \leq i \leq k$.

In order to ensure a fair evaluation, we generated relations of various sizes and characteristics. They differed in: (1) relation size (i.e., the number of tuples), (2) variance in category attribute domain size, and (3) category attribute value skew (see Table 6.2). When the differences in domain sizes were no more than 10% of the average domain size, we took the domain size variance to be low. Otherwise, we took the variance to be high. The distribution of values within a domain was taken to be skewed when 60% of the values were drawn from 40% of the domain. When no skew existed, values were drawn uniformly from the domain. There were 8 category domains and 2 summary domains in all relations. The variations were applied to the category domains only.

In order to evaluate the multi-attribute clustering efficiency, two sets of four tests are performed. Tests $1, 2, 3, 4$ measure the number of disk blocks accessed on average per query for different relation sizes. The four tests correspond to four combinations of relation characteristics: small variance and no data skew, large variance and no data skew, small variance and data

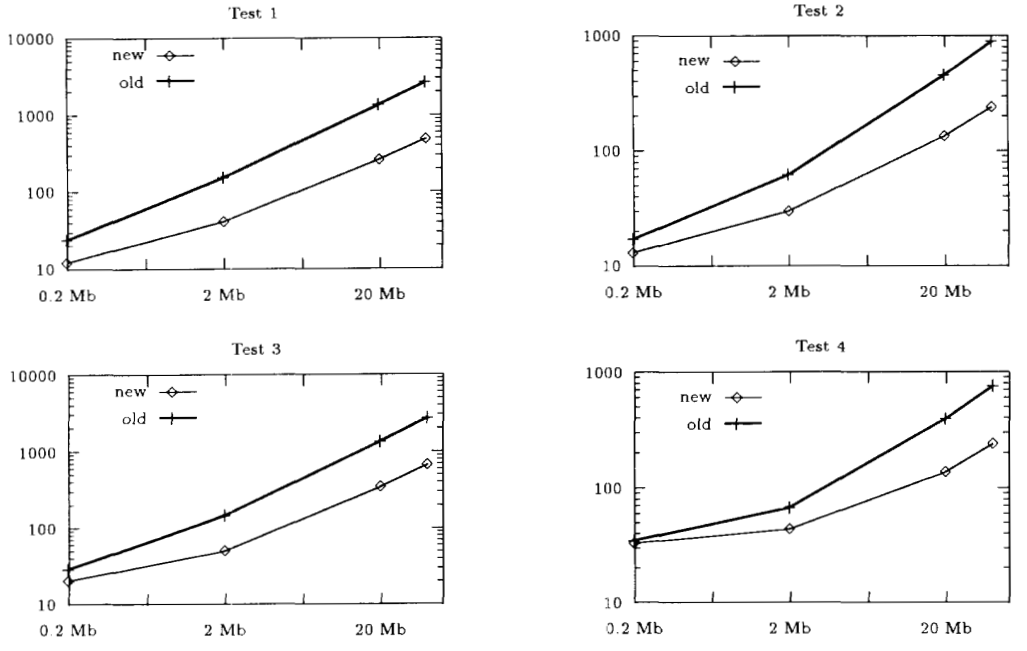

Figure 6.2: Number of data blocks *accessed* versus relation size. The parameters of the tests are found in Table 6.2. Both axes have undergone a $\log_{10}$ transformation so that the units are equally spaced. "new" and "old" stand for relations stored in the TD and conventional storage structures respectively. Observe that the average number of blocks accessed per query is fewer when relations are stored in the TD structure.

skew, large variance and data skew. For each test, a set of 100 selection queries of the form mentioned previously were randomly generated for a given relation size. The number of attributes in the selection criteria of the queries randomly varied between 1 and 4. The total number of blocks accessed was divided by 100 to yield the average number of blocks for that relation size. Four relation sizes were used: 0.2 Mbytes, 2 Mbytes, 20 Mbytes, and 40 Mbytes. The results are shown in Figure 6.2 and the efficiency ratios are in Figure 6.4. The following observations may be made:

- The data blocks accessed on average per query are greatly reduced when the relation is stored via the proposed storage structure. In fact, an efficiency of $1344:262$ $(5.2:1)$ was achieved for test 1 at a relation size of 20 Mbytes.

- The multi-attribute clustering efficiency increases for larger relations. This is seen from the widening gap between each pair of graphs in each test. The efficiency ratios plotted in Figure 6.4 also concur.

- Large variance in attribute domain sizes decreases the multi-attribute clustering efficiency slightly. For example, the efficiency for tests at a relation size of 20 Mbytes under data skew decreases from $1344:262$ $(5.2:1)$ to $458:130$ $(3.5:1)$.

- Data skew increases multi-attribute clustering efficiency because it reduces the variety in the attribute distribution. For instance, the efficiency for tests at a relation size of 20 Mbytes under small domain variances increases from $1363:350$ $(3.9:1)$ with no data skew to $1344:262$ $(5.2:1)$ with data skew.

6.2 Compression efficiency

The second question concerns the compression efficiency of run-length coding on the leading zeroes. The target of interest is the number of data blocks occupied by a relation before and after compression.

To achieve a good mix of relation types, we again varied relation characteristics as above, and performed another set of four tests, numbered $5, 6, 7, 8$. This

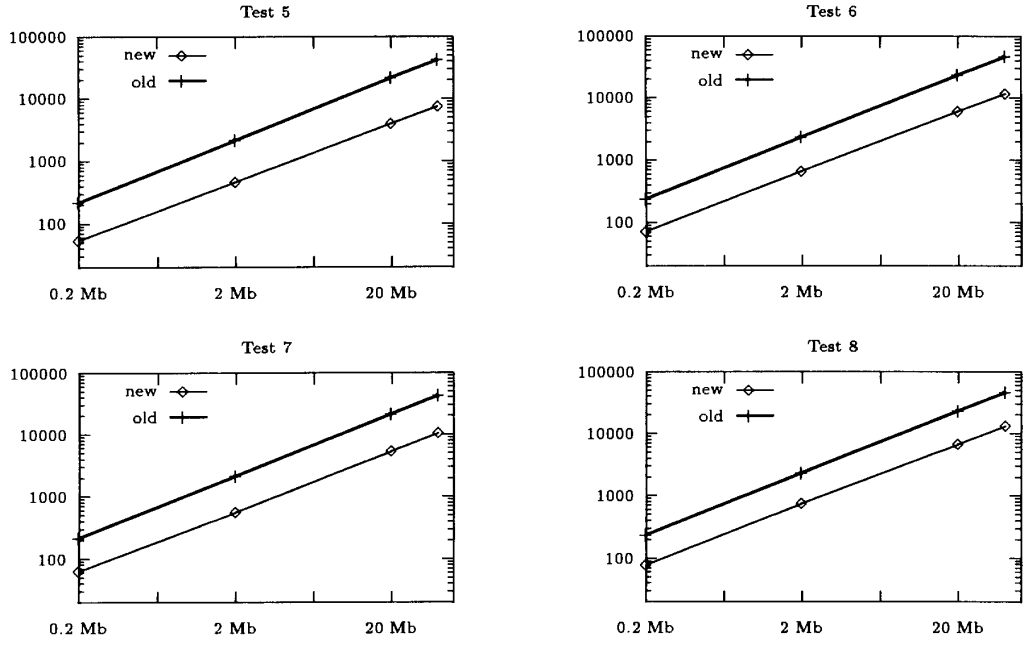

Figure 6.3: Number of data blocks *stored* versus relation size. The parameters of the tests are found in Table 6.2. Both axes have undergone a $\log_{10}$ transformation so that the units are equally spaced. "new" and "old" stand for relations stored in the TD and conventional storage structures respectively. We observe that relations stored under the TD structure are more space efficient than relations under the old structure.

time, however, we measured the number of blocks required by a relation. For each test, we randomly generated a relation and compared its storage requirements before and after storing it in the TD structure. This was performed for various relation sizes. The results are shown in Figure 6.3.

The values obtained are much larger than those in the previous four tests because we are taking the entire relation into account, rather than portions of it. Otherwise, the results of the test are similar to those of previous tests.

As a further illustration of its applicability, we have used the TD structure to store the 1990 census data. The 1990 Public User Microdata Samples (PUMS) from the U.S. Bureau of Census contain records representing 5% or 1% samples of the housing units in the U.S. and of persons residing in them. The 1% samples contain 2.3 million records and occupy 800 Mbytes, while the 5% samples contain 13 million records and occupy 4 gigabytes. Performing aggregational queries on these data is prohibitively slow because of the immense amounts of I/Os generated. By adopting the proposed structure, we are able to improve the response time of statistical queries significantly [13].

In summary, the features afforded by the proposed structure: multi-attribute clustering and compression reduces the amount of I/O incurred during query processing.

7 Related Work

Several techniques have been proposed for physical organization of statistical databases [2, 5, 6, 11] mostly in the context of statistical database compression. Due to space limitations, we shall discuss only the more relevant ones.

Attribute encoding is a popular approach in physical database organization [2, 21]. Since attribute values are often repeated, the set of attribute values occurring in a domain may be mapped to a smaller set of codes to achieve compression. A database may also be *attribute transposed* so that it is stored as a collection of contiguous attribute columns, i.e., all data for an attribute is stored together. In this case,

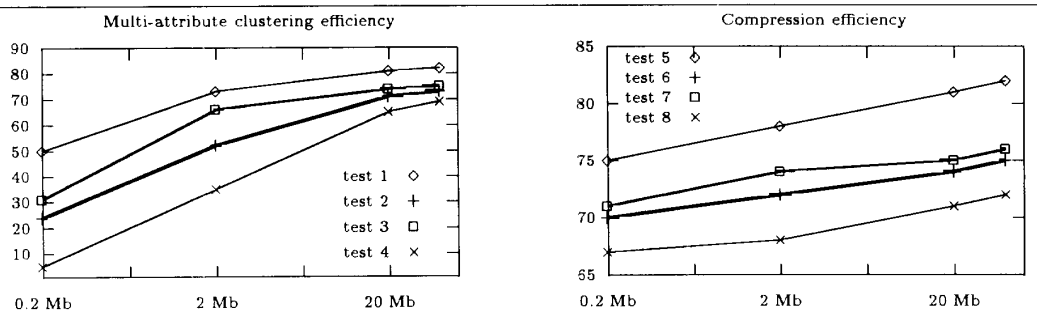

Figure 6.4: Efficiency ratio (in percent) versus relation size. The two sets of graphs are derived from the results of Figure 6.2 and Figure 6.3. Observe that both efficiencies increase with larger relations, which is not entirely apparent in Figure 6.2 and Figure 6.3.

attribute-level coding is useful as it can exploit the numerous repeated occurrence of attribute values. Such schemes have even been carried to the extreme where the database is *bit-transposed*; i.e., all of the data for single bit position of an attribute encoding is stored together [21]. Depending of the type of coding chosen, attribute encoding may also face the problem of running out of codes for new attributes [2].

We have used the attribute transposition technique in our Allegro system[1] for interactive statistical querying of the Public Use Microdata Samples of the U.S. Census of Bureau. However, attribute transposition has both strengths and weaknesses with respect to our tuple-wise storage technique. It is comparatively faster only if the selected set of attributes in a query is very much smaller than the entire set of attributes. In this case, it has the advantage of bringing in only a small subset of the database into memory for processing. However, due to the numerous smaller attribute files generated by attribute transposition, there is a higher level of disk block fragmentation. This may result in higher seek times when locating attributes. In addition, it may no longer be economical to construct indices for each of the attribute files for random access.

[5, 6, 11] are concerned with statistical databases that assume a flat file structure, i.e., a database consisting of one or more sequential file(s) of bytes. Such databases usually contain numeric data generated from the results of laboratory experiments, monitoring of seismic activities, business trends, etc. Such databases exhibit little or no record structure. Hence, they are not useful when tuple structures must be preserved. The primary focus of these techniques is the removal of *constants* from the databases. Constants are runs of identical data values that are usually removed or coded using run-length coding or its variants. As the database is a contiguous sequence of bytes, much of the work is concerned with the determination of efficient mappings between the uncompressed and compressed database. Although statistical databases are relatively static, new records are often inserted as more data are gathered. Such insertions complicate the maintenance of mappings.

[1] Allegro is a proprietary system developed for the Consortium for International Earth Science Information Networking.

8 Conclusions

We have designed a new storage structure for statistical relations that improves the efficiency of statistical queries through multi-attribute clustering and compression, which results in I/O reduction. Our design is motivated by the characteristics of statistical databases:

- Attributes are of two types: category and summary. Category attributes are used as the search keys for access. Summary attributes are the targets of queries, and modifications are rarely made to them, if ever.
- Most queries are access-only in nature; little or no modifications are made. New statistical data that are created as a result of the query do not affect the existing statistical relations.

Our design incorporates these characteristics by conceptually segregating a relation into two halves corresponding to the category and summary portions. Tuples in the category half are stored in the lexicographical order in order to enhance multi-attribute clustering

and permit compression. Since category attributes are generally used as search keys, the new clustering and compression reduce the amount of I/O involved. Because only the summary attributes are modified, tuple modification is performed *in situ*, as in conventional databases.

Much work has been done in improving statistical query processing. The most common approach is the use of precomputed results or summary data. Our approach is not to be seen as orthogonal to existing work as complementing it so as to achieve comprehensive improvements in statistical query processing.

References

[1] S. ABAD-MOTA. Approximate Query Processing with Summary Tables in Statistical Databases. *Lecture Notes in Computer Science*, Vol. 580, pp. 499-515, 1992.

[2] D. S. BATORY. Index Coding: A Compression Technique for Large Statistical Databases. *Proceedings of the Second International Workshop in Statistical Database Management*, pp. 306–314, September 1983.

[3] M. C. CHEN, L. P. MCNAMEE. On the Data Model and Access Method of Summary Data Management. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 4, pp. 519–528, Dec. 1989.

[4] M. C. CHEN, L. P. MCNAMEE, M. MELKANOFF. A Model of Summary Data and its Applications in Statistical Databases. *Proceedings of the 4th International Working Conference on Statistical and Scientific Database Management*, 1988.

[5] S. J. EGGERS, A. SHOSHANI. Efficient Access of Compressed Data. *Proceedings of the International Conference on Very Large Data Bases*, pp. 205–211, 1980.

[6] S. J. EGGERS, F. OLKEN, A. SHOSHANI. A Compression Technique for Large Statistical Databases. *Proceedings of the International Conference on Very Large Data Bases*, pp. 424–434, 1981.

[7] S. P. GHOSH. Statistical Relational Tables for Statistical Database Management. *IEEE Transactions on Software Engineering*, Vol. 12, No. 12, pp. 1106–1116, Dec. 1986. Also published as *IBM Research Report RJ4394*.

[8] S. P. GHOSH. Statistical Relational Model. Chapter 10 in *Statistical and Scientific Databases*, Z. Michalewicz (Editor), Ellis Horwood, New York, 1991.

[9] S. W. GOLOMB. Run-Length Encodings. *IEEE Transactions on Information Theory*, Vol. 12, pp. 399–401, Jul. 1966.

[10] G. HEBRAIL. A Model of Summaries for Very Large Database. *Proceedings of the 3rd International Workshop on Statistical Databases*, 1986.

[11] J. Z. LI, D. ROTEM, H. K. T. WONG. A New Compression Method with Fast Searching on Large Databases. *Proceedings of the International Conference on Very Large Data Bases*, pp. 311–318, 1987.

[12] Z. MICHALEWICZ. *Statistical and Scientific Databases*, Z. Michalewicz (Editor), Ellis Horwood, New York, 1991.

[13] W. K. NG, C. V. RAVISHANKAR. Block Oriented Compression Techniques for Large Statistical Databases. *Under journal review.*

[14] G. OZSOYOGLU, Z. M. OZSOYOGLU, F. MATA. A Language and a Physical Organization Technique for Summary Tables. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 3–16, 1985.

[15] Census of Population and Housing, 1990: Public Use Microdata Samples U.S. (machine readable data files). Prepared by the Bureau of the Census. Washington: The Bureau, 1992.

[16] M. RAFANELLI. Data Models. Chapter 6 in *Statistical and Scientific Databases*, Z. Michalewicz (Editor), Ellis Horwood, New York, 1991.

[17] H. SATO. Statistical Data Models: From a Statistical Table to a Conceptual Approach. Chapter 7 in *Statistical and Scientific Databases*, Z. Michalewicz (Editor), Ellis Horwood, New York, 1991.

[18] A. SHOSHANI. Statistical Databases: Characteristics, Problems, and Some Solutions. *Proceedings of the International Conference on Very Large Data Bases*, pp. 208–222, Sep. 1982.

[19] J. SRIVASTAVA, J. S. E. TAN, V. Y. LUM. TBSAM: An Access Method for Efficient Processing of Statistical Queries. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 4, pp. 414–423, Dec. 1989.

[20] A. U. TANSEL. Statistical Database Query Languages. Chapter 9 in *Statistical and Scientific Databases*, Z. Michalewicz (Editor), Ellis Horwood, New York, 1991.

[21] H. K. T. WONG, H. F. LIU, F. OLKEN, D. ROTEM, L. WONG. Bit Transposed Files. *Proceedings of the International Conference on Very Large Data Bases*, pp. 448-457, 1985.