Inform. Systems Vol. 12, No. 1, pp. 11–15, 1987
Printed in Great Britain

0306-4379/87 $3.00 + 0.00
Pergamon Journals Ltd

APPROXIMATING THE NUMBER OF UNIQUE VALUES OF AN ATTRIBUTE WITHOUT SORTING

MORTON M. ASTRAHAN†
IBM San Jose Research Laboratory, 5600 Cottle Road, San Jose, CA 95193, U.S.A.

and

MARIO SCHKOLNICK‡ and KYU-YOUNG WHANG‡
IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, U.S.A.

(*Received* 30 *November* 1985; *in revised form* 26 *June* 1986)

Abstract—Counts of unique values are frequently needed information in database systems. Especially, they are essential in query optimization and physical database design. Traditionally, exact counts were obtained by sorting, which is an expensive operation. In this paper we present three algorithms for counting unique values by probabilistic methods. These algorithms require only one pass over the data, and produce approximations to the true count with certain standard deviations. For deviations acceptable in practical environments (~10%), the algorithms require only modest amounts of memory space and computation time. We have implemented all three algorithms in System R. We also present the results of the experiments on accuracy and performance of these algorithms.

INTRODUCTION

Many papers have appeared on relational system query optimization, join processing and database design. Most of them involve data models that make use of statistics on the stored information. Typically, these statistics are used for estimating the selectivities of the predicates and the cardinalities of the result relations [1–5]. One of the most commonly required statistics is column cardinality, the number of distinct values stored for a column or attribute. An exact measurement of column cardinality requires sorting, an expensive operation in both instruction executions and I/O operations. Where an index exists, column cardinality can be calculated as a by-product of the sorting required for index creation. Creating and then dropping an otherwise unneeded index would be a convenient way of calculating column cardinality without incurring the cost of maintaining the index. However, we will show that there are cheaper ways of obtaining column cardinality.

In this paper we explore several techniques for obtaining approximations to column cardinality. Since the data models are themselves only approximations, they do not require exact statistics. Each of the techniques is useful under some conditions. Initial measurements show one of them to be most generally advantageous. All of the techniques analyzed here are based on hashing. Hashing has the important property of eliminating duplicates without a need to sort. It also requires only one visit to each element of the set being counted.

In the following discussion of counting algorithms, we will talk about a single column although it is understood that the algorithms can be applied in parallel to more than one column, limited only by the amount of virtual storage allocated for the data structures.

LINEAR COUNTING

Linear counting is a straightforward application of hash transformation. It requires a bit map which is initially set to all zeros. The relation being counted is scanned (i.e. tuples are accessed one by one) and a hash function maps each data value of the column into a position in the bit map. The addressed bit is set to "1". Figure 1 illustrates this process for one column of a four-tuple relation. If there were no hash collisions (different data values mapping to the same bit position), the final count of "1" bits would be the desired column cardinality. In Fig. 1, this count is three.

For a given bit map size y, we can calculate the expected number of collisions that will occur when we hash a set of elements whose cardinality is x. Assuming that distinct values are transformed by the hash function randomly over the range of hash values, the expected number of distinct hash values, $E(z')$, is given by

$$E(z') = y \times [1 - (1 - 1/y)^x], \qquad (1)$$

where y is the number of bits in the map [5]. Thus, having measured z', the number of "1"s in the map

†Current address: 2896 Gardendale Dr., San Jose, CA 95125, U.S.A.

‡Most work was done while the authors were at IBM San Jose Research Laboratory.

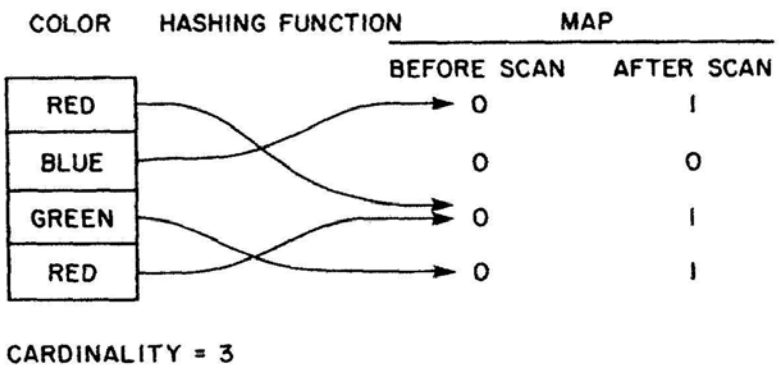

Fig. 1. Linear counting.

after the scan, we can intuitively estimate x as

$$x' = -y \times \ln(1 - z'/y), \tag{2}$$

assuming $y \gg 1$, where x' is the random variable representing the estimation of x.

The expected accuracy of the estimation depends on the map size and on the ratio of column cardinality to map size, called the loading factor. An analysis shows that for a loading factor less than one, the standard error is less than $0.85/\sqrt{y}$ [6] The standard error is defined as the standard deviation of the random variable representing the estimation of the column cardinality divided by its true value. For example, for a map size of 100 bits and a loading factor of one, we can expect a standard error of about 8.5%. Note that the expected error is inversely proportional to the square root of the map size, becoming very small for large maps.

Linear counting performs very well when the cardinality of the set being measured is not extremely big. In practice, for reasonable accuracy over a large range of relation sizes, the loading factor should be kept below two. In order to ensure this, we must use a map having at least half as many bits as the number of tuples in the relation. For a 200K-tuple relation, this means each map requires approximately three pages of 4K bytes. If the map pages do not all fit in the system's available real memory, then each page must be fetched into the memory for each tuple of the relation. Thus, the size of available real memory limits the number of columns that can be processed simultaneously. For a relation cardinality over 20 million, the storage required can become impractical for even a single column.

LOGARITHMIC COUNTING

Logarithmic counting is described in reports by Flajolet and Martin [7, 8]. (In the references it is called probabilistic counting). The hash space y is made sufficiently large so that the number of collisions is expected to be negligible. We found a space of size 2^{31} to be convenient. However, one does not keep a hash table of this size. Rather, a 32-bit word is used to capture statistical information about the result of the hashing process. This 32-bit word is called a map. For each data value, a hash value is calculated, as illustrated in Fig. 2. For convenience, only the leftmost 3 bits are shown. Each has value is transformed by a function which leaves only the leftmost "1" bit unchanged, all other bits being set to "0". The transformed values are OR'd into the map. After the scan, the average length n of the leftmost string of uninterrupted "1's" for each column is proportional to the logarithm of the desired count. The referenced analysis shows that if n is the position of the leftmost "0" in the map after the scan, starting with position 0 on the left, then the column cardinality can be estimated as

$$\text{Cardinality} = 2^n/0.7735. \tag{3}$$

In the illustration, the actual cardinality is 6 and the measured value is 5.2.

One can get an intuitive understanding of this by considering the leftmost bits of the map. The leftmost bit will be "1" only if the leftmost bit of one of the hash values is "1". This ought to happen in every second value on the average, so the count of distinct values is at least 2^1. The pattern "11" occurs at the leftmost bits of a map only if the patterns "01 . . ." and "1 . . ." occurred as the left-hand bits of hash values. The pattern "01 . . ." would be expected once in every 4 values. If the final pattern of the leftmost bits is "110" as illustrated, then "001" did not occur as the left-hand bits of any hash value, implying that the count is likely to be less than 2^3.

An error analysis [8] shows that, with a single map, we expect the standard error of the estimated count to be 0.78. This means that we expect the estimated count to be within 78% of the true count about 67% of the time, assuming a normal distribution. However, the error can be reduced by using m maps, with a different hash function for each. By taking n as the average value obtained from m independent maps, the standard error becomes $0.78/\sqrt{m}$. By using 64 maps we can reduce the expected standard error from 78% to 9.8%. There are several ways to achieve this benefit without having to use m completely different

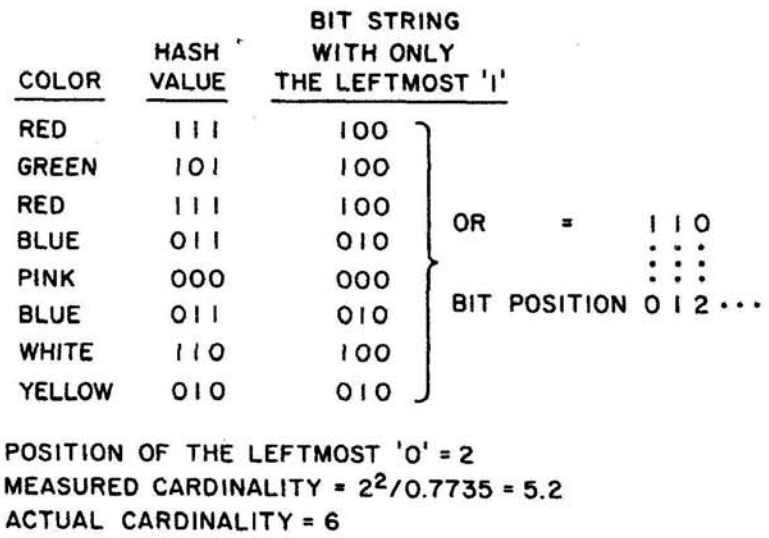

Fig. 2. Logarithmic counting.

hash functions. Flajolet and Martin [7] use part of the hashed value to select one of m maps and then update that map using the remainder of the hashed value. Whang uses a table of m random numbers. Each number is XOR'd with the hashed value and the result is used to update the corresponding map.

Logarithmic counting uses much less virtual storage than linear counting but requires more computation. It is suited to relations having more tuples or columns than can be conveniently represented by the virtual storage bit maps of linear counting.

SAMPLE COUNTING

Sample counting is based upon a suggestion by Wegman [9]. As with the previous methods, the relation is scanned and a hash function is applied to each data value. As illustrated in Fig. 3, a list is maintained of the hash values already encountered. Subsequent values are compared with those in the list and are put into the list if not already there. However, only a fraction of the hash values are eligible to be entered into the list. They are chosen by means of sampling parameter K and a sample reference pattern. The values chosen must have K bits that exactly match the corresponding K bits of the reference pattern. In the illustration, K is 3 and only those values whose rightmost 3 bits are "1"s are chosen. The sampling fraction is thus $1/2^K$, and the cardinality is found by multiplying the final length of the list by 2^K. Note that there is a granularity in the calcualted cardinality since it must be an integer multiple of 2^K.

The list space must be large enough to hold $1/2^K$ of the actual column cardinality. However, since the cardinality is what we are calculating, it is not known in advance. We therefore start with an initial length L for the list. We start with $K = 0$, counting all values. When the list is full, we increase K and purge the list of all values that fail the more stringent test. If we increase K by 1, we expect to remove half the entries. When all the tuples have been tested, the entries in each list are exactly what they would have

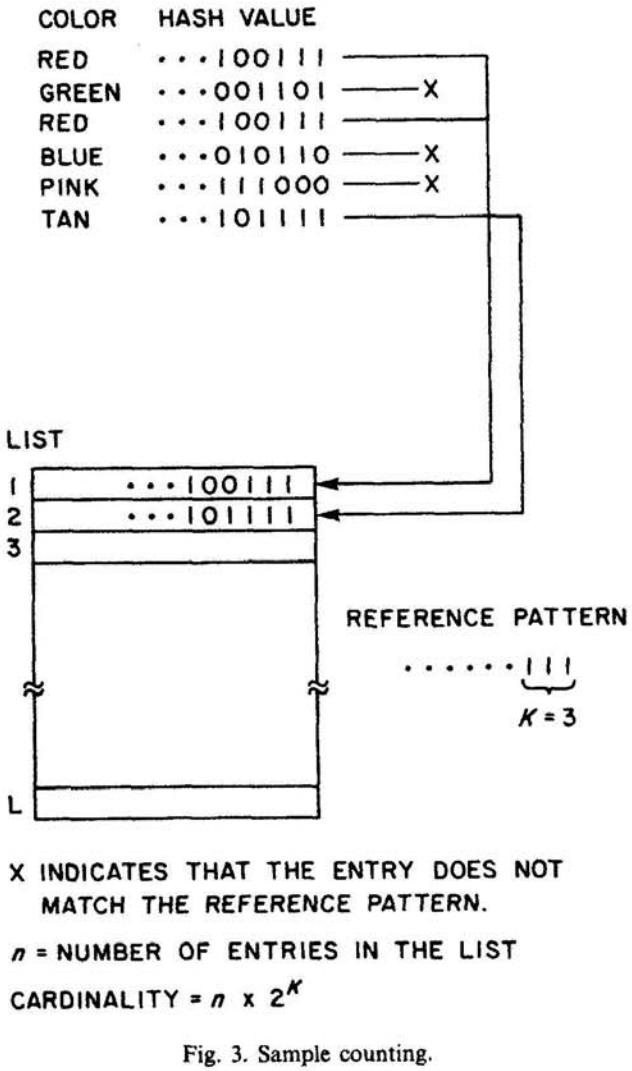

Fig. 3. Sample counting.

been if we had started processing each column with its final value of K. Analysis shows that, when the sampling fraction is much less than 1 (i.e. $K \gg 1$), we can expect a standard error of $1.2/\sqrt{L}$ [10]. For a 100-element list, this is about 12%.

As with linear and logarithmic countings, we can compensate for hash collisions by using equation (2) with z as the measured count and y as the number of possible hash values. The correction is less than 1% if $z/y < 0.02$. Hence, when y is $2^{31} - 1$, no compensation is needed until $z \geqslant 2^{26}$.

HASHING

All the counting algorithms examined require an efficient hashing function that can be applied to all data types encountered. In System R environment these are the SQL data types (binary, decimal, float, fixed and variable character string). We choose a hashing function that is a member of the Universal-2 class of functions [11]. The basic algorithm is to take two $3\frac{1}{2}$-byte pieces of the input string at a time, add a random number to each piece, and then multiply the two pieces together. The resultant product is then taken modulo $2^{31} - 1$, and added to the previously generated hash value. The sum is then taken modulo $2^{31} - 1$. Ideally, an entire class of hashing functions should be used with an independent list for each. However, through all the tests we have made, the performance of this single function has proved to be satisfactory.

MEASUREMENTS

The authors implemented all three counting algorithms in system R from 1981 to 1982. Whang's version of the logarithmic counting was used. We made accuracy measurements for linear and logarithmic counting, using five columns of a 2053-tuple and of a 24,359-tuple relation. For linear counting, with a loading factor less than one (thus, the memory space used for the data structures was 257 bytes/column and 3045 bytes/column, respectively), the errors averaged over the five columns were 1.0% and 0.4% of the true count, respectively. Logarithmic counting, with 64 maps (256 bytes/column) for each column, resulted in average errors of 10.7% and 11.6%.

Two relations were generated to test the accuracy of sample counting. Relations A and B had a cardinality of 100K and 7 columns each. All the basic data types were represented among the columns, but the relations differed in their column cardinalities. Relation A had a key column (cardinality 100K) and an average cardinality of 74K over the other 6 columns. Relation B also had a key column and had an average cardinality of 10K over the other 6 columns. All measurements were made in the environment of System R and VM370 on an IBM 3033. Performance was measured by Virtual CPU time (VCPU). Using relations A and B and a list size of 100 (400 bytes/column), sample counting showed errors averaged over the 7 columns of 4.1% and 5.2% respectively. Notice that the errors obtained were rather favorable compared with the prediction because the reference pattern chosen turned out to be favorable to the key sets used for the experiments.

CONCLUSIONS

We have presented three algorithms for counting unique values by probablistic methods. All three algorithms show error bounds acceptable in practical environments ($\sim 10\%$), while using a modest amount of memory space and computation time. Linear counting, which is the simplest, provides the highest accuracy of all, but the performance degrades for extremely large relations. Logarithmic counting proves to be fairly accurate without being limited by the sizes of relations. We have found, however, it is somewhat inaccurate when the column cardinality is very small (e.g. less than 30). This problem occurs because some approximations used in the analysis are no longer valid when the column cardinality is too small. Sample counting provides the best result, satisfying all the requirements: memory storage space, performance, and accuracy. In the measurement, Sample counting took 14 sec plus 3 sec per column to process a 100K-tuple relation with 6 columns (Relation B). This result indicates that sample counting is much less costly compared with the method of creating indexes. Creating indexes would have cost 114 sec to process the same relation.

All three algorithms have been implemented running successfullly in System R. We believe this paper provides many designers of database management systems with a valuable new approach to obtaining column cardinalities.

Acknowledgements—The authors wish to thank Glen Foster, Nigel Martin, Philippe Flajolet and Mark Wegman for their contributions at various stages of the project. The authors are grateful for thoughtful comments from Shel Finkelstein on an earlier version of this paper.

REFERENCES

[1] M. Hammer and A. Chan, Index selection in a self-adaptive database management system. In *Proc. Int. Conf. on Management of Data*, Washington, D.C., *ACM SIGMOD* pp. 1–8 (1976).

[2] R. Krishnamurthy and S. Morgan, Query processing on personal computers: a pragmatic approach. In *Proc. 10th Int. Conf. Very Large Data Bases*, Singapore (1984).

[3] M. Schkolnick and P. Tiberio, Estimating the cost of updates in a relational database, *Ass. comput. Mach. Trans. Database Systems* **10**(2), 959–965 (1985).

[4] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and I. G. Price. Access path selection in a relational database management system. In *Proc. Int. Conf. on Management of Data*, Boston, MA., *ACM SIGMOD* pp. 23–24 (1979).

[5] K. Whang, G. Wiederhold and D. Sagalowicz, Estimating block accesses in database organizations—a closed noniterative formula. *Commun. Ass. comput. Mach.* **26**(1), 940–944 (1983).
[6] K. Whang, H. Taylor and B. Vander-Zanden, Linear counting: a detailed analysis. In preparation.
[7] P. Flajolet and N. Martin, *Probabilistic counting*. In *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pp. 76–82 (1983).
[8] P. Flajolet and N. Martin, *Probabilistic Counting for Database Applications*, Rapports de Recherche No. 313, INRIA, Le Chesnay, France (1984).
[9] M. Wegman, Sample counting, Private communication (December 1983).
[10] P. Flajolet, On Wegman's adaptive sampling algorithm, draft (1984).
[11] J. Carter and M. Wegman, Universal class of hash functions. *J. Comput. Systems Sci.* **18**(2), 143–154 (1979).