Estimating the Compression Fraction of an Index using Sampling

Stratos Idreos[#1], Raghav Kaushik[*2], Vivek Narasayya[*3], Ravishankar Ramamurthy[*4]

[#]*CWI Amsterdam*
[1]idreos@cwi.nl
[*]*Microsoft Corp, USA*
{[2]skaushi, [3]viveknar, [4]ravirama} @microsoft.com

Abstract—**Data compression techniques such as null suppression and dictionary compression are commonly used in today's database systems. In order to effectively leverage compression, it is necessary to have the ability to efficiently and accurately estimate the size of an index if it were to be compressed. Such an analysis is critical if automated physical design tools are to be extended to handle compression. Several database systems today provide estimators for this problem based on random sampling. While this approach is efficient, there is no previous work that analyses its accuracy. In this paper, we analyse the problem of estimating the compressed size of an index from the point of view of worst-case guarantees. We show that the simple estimator implemented by several database systems has several "good" cases even though the estimator itself is agnostic to the internals of the specific compression algorithm.**

I. Introduction

Data compression is commonly used in modern database systems. Compression can be utilized in database systems for different reasons including: 1) Reducing storage/archival costs, which is particularly important for large data warehouses 2) Improving query workload performance by reducing the I/O costs 3) Reducing manageability costs by reducing the time taken and storage costs for backup, recovery and log shipping.

While data compression does yield significant benefits in the form of reduced storage costs and reduced I/O there is a substantial CPU cost to be paid in decompressing the data. Thus the decision as to when to use compression needs to be taken judiciously.

Given that compression increases the space of physical design options, there is a natural motivation to extend automated physical design tools (see [9] for an overview) to handle compression. Such tools take as input a query workload and a storage bound to produce a set of indexes that can fit the storage bound while minimizing the cost of the workload. In order to meet the storage bound as well as reason about the I/O costs of query execution, it is necessary to perform a quantitative analysis of the effects of compression:

1) Given an index, how much space will be saved by compressing it?
2) Given a workload, how is its performance impacted by compressing a set of indexes?

One of the key challenges in answering the above questions is to estimate the size of an index if it were to be compressed. Since the space of physical design options is large, it is important to be able to perform this estimation *accurately and efficiently*. The naïve method of actually building and compressing the index in order to estimate its size, while highly accurate is prohibitively inefficient.

Thus, we need to be able to accurately estimate the compressed size of an index *without* incurring the cost of actually compressing it. This problem is challenging because the size of the compressed index can depend significantly on the data distribution as well as the compression technique used. This is in contrast with the estimation of the size of an uncompressed index in physical database design tools which can be derived in a straightforward manner from the schema (which defines the size of the corresponding column) and the number of rows in the table.

Besides physical database design, such analysis can also be leveraged for other applications such as capacity planning, for example to estimate the amount of storage space required for data archival. We also note that even though compression can be invoked on both tables and indexes, in this paper we primarily focus on indexes (clustered and non-clustered). Our results can be extended in a straightforward manner for the case of tables.

Random sampling is a well-known approach to yield *efficient* estimates for various database statistics [4]. Some database systems today (e.g., [10][12]) leverage sampling for estimating the compressed size of an index. The key idea is to draw a random sample and simply return the compression ratio obtained for the sample as an estimate of the true compression ratio. The advantages of this approach include the simplicity of the algorithm and the fact that it is agnostic to the internals of the underlying compression technique. While this estimator is indeed efficient, there is no previous work (either analytical or empirical) that studies its accuracy.

In this paper, we focus on the problem of estimating the *compression fraction*, defined as the ratio of the size of the compressed index to the size of the uncompressed index. We conduct our analysis for two commonly used compression techniques - null suppression and dictionary compression (we review these techniques in Section II). We examine the worst-case guarantees (Section III) that can be provided by estimators that leverage sampling (we call the estimator *SampleCF*) for the above compression techniques.

One of the main contributions of this paper is to show that there are many "good" cases for *SampleCF*, even though the estimator is agnostic to the internals of the compression algorithm. We first show that for null suppression, *SampleCF*

is an unbiased estimator with low variance. For the case of dictionary compression, we find that the problem of estimating the compressed fraction is closely related to the problem of estimating the number of distinct values using sampling which is known to be hard [1]. Despite this connection, we show that many "good" cases exist for *SampleCF*. We summarize our results in Section IV.

II. PRELIMINARIES

A. An Overview of Compression Techniques

Compression techniques have been well studied in the context of database systems (see [7][8] for an overview). While a variety of techniques have been explored by the research community, in this paper we focus on two compression techniques that are commonly used in databases today which we briefly review below.

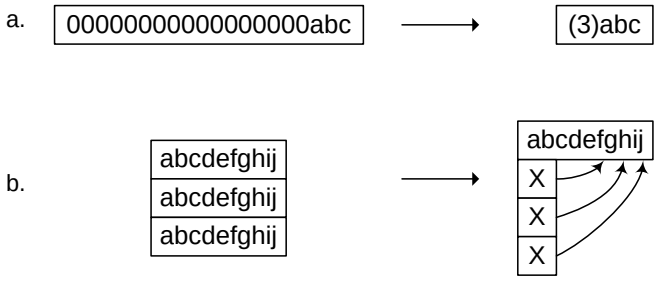

Fig 1. Compression Techniques
a: Null Suppression b: Dictionary Compression

Null Suppression (NS): This technique is used to suppress either zeros or blanks in each tuple. The key idea is to represent a sequence of zeros and blanks by a special character, followed by a number that indicates the length of the sequence. For example, consider a single column index whose data type is declared as CHAR(20). Consider the value 'abc'. If this is stored in an uncompressed fashion, this would use all 20 bytes, while null suppression would only store the value 'abc' along with its length, in this instance 3 bytes (see Figure 1.a). In the case of multi-column indexes, each column is compressed independently.

Dictionary Compression: This technique takes as input a set of tuples and replaces the actual values with smaller pointers. The mapping between the distinct values and the pointers is maintained in a dictionary. Consider the set of tuples with the data value 'abcdefghij' in Figure 1.b. Dictionary compression stores the data value once and replaces each of the actual occurrences with a pointer. In practice, in order to minimize the overhead of looking up the dictionary, commercial systems typically apply this technique at a page level and the dictionary is maintained inline in every page. This ensures that the dictionary lookup does not need additional I/Os. In the case of multi-column indexes, each column is compressed independently.

B. Compression Fraction

We evaluate the effectiveness of a compression technique (see [8]) by using a metric *compression fraction* (CF) which is defined as follows.

$$CF = \frac{DataSize\ With\ Compression}{DataSize\ Without\ Compression}$$

The *CF* as defined is a value between 0 and 1 (if we ignore degenerate cases where compression can actually increase the size of the original index). A lower compression fraction corresponds to a higher reduction in the size.

C. Estimating Compression Fraction Using Sampling

While the *CF* for an index can be computed accurately by actually compressing the index, this is prohibitively inefficient (especially for large data sets). Ideally, we need to be able to estimate the *CF* both *accurately* and *efficiently*, specifically without incurring the cost of actually compressing the entire index.

Some database systems today (e.g., [12][10]) leverage random sampling in order to provide a quick estimate of the *CF*. The idea (see Figure 2) essentially is to randomly sample a set of tuples from the table, build an index on the sample, compress it and return the compression fraction obtained in the sample as an estimate of the compression fraction of the entire index. The main advantages of this approach are as follows: 1) the algorithm is simple to implement 2) it is agnostic to the actual compression technique used and thus requires no modification when we incorporate a new compression technique. We note that if the (uncompressed) index already exists, we can obtain the random sample more efficiently from the index instead of the base table.

Algorithm SampleCF (*T*, *f*, S, C)
// Table T
// Sampling fraction f
// Sequence of Columns in the index S
// Compression Algorithm C

1. T' = uniform random sample of $f \times T$ rows from T
2. Build index I'(S) on T'
3. Compress index I' using C
4. Return *CF* for index I'

Fig 2. Estimating CF using Sampling

While sampling is efficient, there is no previous work to our knowledge that studies its accuracy for estimating the compression fraction. As noted in Section I, analyzing the accuracy is the goal of this paper. We evaluate an estimator for the compression fraction (*CF'*) by using the ratio error which is defined as follows.

$$RatioError = MAX(\frac{CF}{CF'}, \frac{CF'}{CF})$$

In this paper, we assume that the algorithm outlined in Figure 2 uses uniform random sampling over all *tuples* with

replacement. We do note that in contrast, commercial systems typically leverage *block-level* sampling (in which all the rows from a randomly sampled page are included in the sample). Our analysis for uniform tuple sampling is still a useful starting point to understand the guarantees. Extending the analysis to account for page sampling is part of future work.

III. Analysis of SampleCF Estimator

In this section, we analyze the accuracy of the sampling based estimator *SampleCF* introduced in Section II. For the purpose of analysis in the rest of the paper, we assume a table T that has a single column A which is a character field of *k* bytes (i.e. char(*k*)). In the rest of the paper, we use the terms "*CF* of table T" and "*CF* of column A" both to refer to the *CF* of an index on A. We note that our analysis extends for the case of multi-column indexes in a straightforward manner.

We assume that the size of any individual tuple *k* cannot exceed the page size used by the database system. Let the number of rows in the table be *n* and the number of distinct values be *d*. We define the null-suppressed length (in bytes) of a tuple as its *actual length*, denoted l_i. We assume a uniform random sample with replacement of size *r* rows from the table. The number of distinct values of A in the sample is denoted *d'*. This notation is summarized in Table 1 below.

TABLE I
Notation Used in Analytical Model

n	Number of rows in the table
d	Number of distinct values in the table
D	Set of distinct values in the table
k	Size of each tuple
l_i	Null suppressed length of each tuple
r	Number of rows in the sample
d'	Number of distinct values in the sample

We study the expected value of the estimate, its variance and the worst-case guarantees. As noted in Section II, to our knowledge, no prior work has characterized the accuracy of estimating the compression fraction. We organize the discussion in this section by the compression method.

A. *Null Suppression*

We first study Null Suppression. The original size of the table is $n \times k$, since each tuple in its uncompressed form uses *k* bytes (recall that the table has a single column field of type char(*k*)). When we use null suppression, we get rid of any unnecessary blanks and only store the actual length of the tuple (l_i) but we also need to keep track of the length which requires $\log k$ bytes. Thus, the compression fraction for null suppression is given by the following expression.

$$CF_{NS} = \frac{n \times \log k + \sum_1^n l_i}{n \times k}$$

In this expression, the only unknown is $\sum_1^n l_i$. Thus, the problem of estimating the compression fraction for Null Suppression reduces to the problem of estimating this sum. The usage of random sampling for estimating a *sum* aggregation has been studied in prior work [2]. Specifically, drawing a random sample, computing the sum over the sample and scaling it up is known to be unbiased. The estimate returned by *SampleCF* is:

$$\mathrm{CF'_{NS}} = \frac{r \times \log k + \sum_1^r l_j}{r \times k} = \frac{n \times \log k + \frac{n}{r}\sum_1^r l_j}{n \times k}$$

We can observe that in computing $\mathrm{CF'_{NS}}$, we have performed the same scaling. Thus $\mathrm{CF'_{NS}}$ is an unbiased estimate of CF. Sampling based estimation of *sum* is however known to suffer from potentially large variance [2]. However, in our setting, the length of the tuples is bounded by *k*. This translates to corresponding bounds on the variance of $\mathrm{CF'_{NS}}$. We formalize this intuition in the following result.

Theorem 1: *Consider a table T with a single column of type char(k), and $n \geq k$ rows. The estimate* $\mathrm{CF'_{NS}}$ *is unbiased, that is $E[CF'_{NS}] = CF$, and its standard deviation can be bounded as: $\sigma(CF'_{NS}) \leq \frac{1}{f \times \sqrt{r}}$ where f = r/n is the sampling fraction.*

We illustrate the implication of this result using an example.

Example 1. Suppose that table *T* has *n = 100 million* rows. Suppose that we draw a sample of size *r = 1 million* (which corresponds to a 1% sample). Then, Theorem 1 implies that the standard deviation of $\mathrm{CF'_{NS}}$ is at most $\frac{100}{\sqrt{1000000}} = 0.1$. □

B. *Dictionary Compression*

When we use dictionary compression (see Figure 1.b), for a set of identical values in a page, we store the original value in the dictionary and store a pointer to this value instead (which in general requires $\mathrm{O}(\log d)$ bytes). Let *p* denote the size of the pointer in bytes. As mentioned in Section II-A, the dictionary is typically in-lined in each page. For each distinct value *i*, let *Pg(i)* denote the number of pages that this value occurs in when compressed. We note that each distinct value is stored once in each of the *Pg(i)* pages. The following expression denotes the compression fraction of Dictionary Compression (note that the summation is over the distinct values in T):

$$CF_{DC} = \frac{n \times p + \sum_{i \in D} k \times Pg(i)}{n \times k}$$

In order to simplify the analysis and isolate the effects of each of the above factors (pointers per occurrence and paging), we consider a simplified model of dictionary compression in which the paging effects are ignored. Here, dictionary compression stores a "global" dictionary in which each distinct value is stored once and each row has a pointer to the dictionary. Under the simplified model, the compression fraction of Dictionary Compression is:

$$CF_{DC} = \frac{n \times p + d \times k}{k \times n}$$

We note that for the above expression the only unknown is the number of distinct values (d). There is no known unbiased distinct value estimator that works off a random sample. In fact, prior work (e.g., [1]) has shown that any estimator that uses uniform random sampling for distinct value estimation must yield a significant ratio error in the worst case. In spite of this fact, we now show that the estimator *SampleCF* yields an estimate that has bounded error in several cases. Recall that the estimate yielded by *SampleCF* is captured by the following expression.

$$CF'_{DC} = \frac{r \times p + d' \times k}{k \times r}$$

We separate the analysis into two cases – where the number of distinct values is "small" and "large". When the number of distinct values is "small", the $n \times p$ factor in the expression for CF_{DC} can dominate the other term which involves d and as a result we can still obtain an accurate estimate. This intuition is formalized in the following result.

Theorem 2: *Fix constants ϵ, c, f and a function $g: N \rightarrow N$(where N stands for the set of natural numbers) such that g is $o(n)$. For any n that is sufficiently large , for any table T with n rows, $d = g(n)$ distinct values, and column length $c \log d \geq k \geq \log d$ the following holds. If we run SampleCF with $r = fn$, then the expected ratio error of CF'_{DC} is at most* $1 + \epsilon$.

Now we consider the case where the number of distinct values is large. We demonstrate that *SampleCF* yields a bounded ratio error estimate when the number of distinct values is "large". Intuitively, if the number of distinct values in T is "large", we can show that the fraction of distinct values in the sample will also be significant. This implies that both d' and r are also proportional to n, which further implies that we can obtain a bound on the ratio error.

Theorem 3: *Fix constants α, f. For any n that is sufficiently large, for any table T with n rows and $d \geq \alpha \times n$ distinct values, the following holds. If we run SampleCF with $r \geq fn$, then the expected ratio error of CF'_{DC} is at most* $\max\left(\frac{1}{\alpha}, \frac{f}{1-e^{-f}}\right)$.

Thus, despite the fact that estimating the compression fraction for dictionary compression is related to the problem of distinct value estimation, we are able to show (for a simplified model of dictionary compression) that many cases exist where we can bound the ratio error. Our experimental results (omitted due to lack of space) also confirm that the *SampleCF* algorithm can be an effective estimator in practice for the case of both null suppression and dictionary compression.

IV. Conclusions

In this paper, we identified the problem of estimating the compression fraction using uniform random sampling, which is a measure of how much a given index gets compressed. We analyzed the estimation accuracy for two popular compression techniques. Our results are summarized in Table 2.

TABLE II
Summary of Results

Compression Technique	*Estimator*	*Bias*	*Small d (o(n))*	*Large d (O(n))*
Null Suppression	*SampleCF*	No	Variance at most $\frac{1}{f \times \sqrt{r}}$	Variance at most $\frac{1}{f \times \sqrt{r}}$
Dictionary Compression	*SampleCF*	Yes	Expected ratio error close to 1	Expected ratio error at most constant

We found that a simple estimator *SampleCF* that draws a uniform random sample and returns the compression fraction on the sample as its estimate has low error for many cases. It is interesting future work to extend our analysis to model paging effects in dictionary compression as well as consider block-level sampling.

References

[1] M.Charikar, S.Chaudhuri, R.Motwani, V.Narasayya. Towards Estimation Error Guarantees for Distinct Values. In Proceedings of PODS 2000.

[2] S.Chauduri et.al. Overcoming Limitations of Sampling for Aggregation Queries. In Proceedings of ICDE 2001.

[3] G.Graefe, L.Shapiro. Database Compression and Database Performance. In Symp. On Applied Computing. 1991.

[4] F.Olken, D.Rotem. Random Sampling from Databases. A Survey. Statistics and Computing. March 1995.Vol 5.

[5] J.S. Vitter. Random Sampling with a Reservoir. ACM Transactions on Math. Software. 11(1): 37-57 (1985)

[6] M.Poess, D.Potapov. Data Compression in Oracle. In Proceedings of VLDB 2003.

[7] M.Roth, Scott J. VanHorn. Database Compression. Sigmod Record 22(3). 1993.

[8] D.G.Severance. A Practitioner's Guide to database compression-tutorial. Inf. Sys. 8(1). 1983.

[9] Special Issue on Self-Managing Database Systems. IEEE Data Engineering Bulletin. Volume 29, Number 3, 2006.

[10] Oracle Advanced Compression. White Paper. http://www.oracle.com/

[11] IBM DB2 Data Compression. http://www.ibm.com/software/data/db2/compression

[12] SQL Server Data Compression. https://blogs.msdn.com/sqlserverstorageengine/