HHS Public Access

Author manuscript

KDD. Author manuscript; available in PMC 2017 February 06.

Published in final edited form as:

KDD. 2016 August ; 2016: 1575–1584. doi:10.1145/2939672.2939867.

Squish: Near-Optimal Compression for Archival of Relational Datasets

Yihan Gao and
University of Illinois at Urbana-Champaign, Urbana, Illinois

Aditya Parameswaran
University of Illinois at Urbana-Champaign, Urbana, Illinois

Abstract

Relational datasets are being generated at an alarmingly rapid rate across organizations and industries. Compressing these datasets could significantly reduce storage and archival costs. Traditional compression algorithms, e.g., gzip, are suboptimal for compressing relational datasets since they ignore the table structure and relationships between attributes.

We study compression algorithms that leverage the relational structure to compress datasets to a much greater extent. We develop Squish, a system that uses a combination of Bayesian Networks and Arithmetic Coding to capture multiple kinds of dependencies among attributes and achieve near-entropy compression rate. Squish also supports user-defined attributes: users can instantiate new data types by simply implementing five functions for a new class interface. We prove the asymptotic optimality of our compression algorithm and conduct experiments to show the effectiveness of our system: Squish achieves a reduction of over 50% in storage size relative to systems developed in prior work on a variety of real datasets.

1. INTRODUCTION

From social media interactions, commercial transactions, to scientific observations and the internet-of-things, relational datasets are being generated at an alarming rate. With these datasets, that are *either costly or impossible to regenerate, there is a need for periodic archival,* for a variety of purposes, including long-term analysis or machine learning, historical or legal factors, or public or private access over a network. Thus, despite the declining costs of storage, compression of relational datasets is still important, and will stay important in the coming future.

One may wonder if compression of datasets is a solved problem. Indeed, there has been a variety of robust algorithms like Lempel-Ziv [21], WAH [20], and CTW [17] developed and used widely for data compression. However, these algorithms do not exploit the relational

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

structure of the datasets: attributes are often correlated or dependent on each other, and identifying and exploiting such correlations can lead to significant reductions in storage. In fact, there are many types of dependencies between attributes in a relational dataset. For example, attributes could be functionally dependent on other attributes [3], or the dataset could consist of several clusters of tuples, such that all the tuples within each cluster are similar to each other [8]. The skewness of numerical attributes is another important source of redundancy that is overlooked by algorithms like Lempel-Ziv [21]: by designing encoding schemes based on the distribution of attributes, we can achieve much better compression rate than storing the attributes using binary/float number format.

There has been some limited work on compression of relational datasets, all in the recent past [3, 6, 8, 12]. In contrast with this line of prior work, Squish uses a combination of Bayesian Networks coupled with Arithmetic Coding [19]. Arithmetic Coding is a coding scheme designed for sequence of characters. It requires an order among characters and probability distributions of characters conditioned on all preceding ones. Incidentally, Bayesian Networks fulfill both requirements: the acyclic property of Bayesian Network provides us an order (i.e., the topological order), and the conditional probability distributions are also specified in the model. Therefore, Bayesian Networks and Arithmetic Coding are a perfect fit for relational dataset compression.

However, there are several challenges in using Bayesian Networks and Arithmetic Coding for compression. First, we need to identify a new objective function for learning a Bayesian Network, since conventional objectives like Bayesian Information Criterion [15] are not designed to minimize the size of the compressed dataset. Another challenge is to design a mechanism to support attributes with an infinite range (e.g., numerical and string attributes), since Arithmetic Coding assumes a finite alphabet for symbols, and therefore cannot be applied to those attributes. To be applicable to the wide variety of real-world datasets, it is essential to be able to handle numbers and strings.

We deal with these challenges in developing Squish. As we show in this paper, the compression rate of Squish is near-optimal for all datasets that can be efficiently described using a Bayesian Network. This theoretical optimality reflects in our experiments as well: Squish *achieves a reduction in storage on real datasets of over 50% compared to the nearest competitor*. The reason behind this significant improvement is that most prior papers use suboptimal techniques for compression.

In addition to being more effective at compression of relational datasets than prior work, Squish is also more *powerful*. To demonstrate that, we identify the following desiderata for a relational dataset compression system:

- *Attribute Correlations (AC).* A relational dataset compression system must be able to capture correlations across attributes.
- *Lossless and Lossy Compression (LC).* A relational dataset compression system must be general enough to admit a user-specified error tolerance, and be able to generate a compressed dataset in a lossy fashion, while respecting the error tolerance, further saving on storage.

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

- *Numerical Attributes (NA).* In addition to string and categorical attributes, a relational dataset compression system must be able to capture numerical attributes, that are especially common in scientific datasets.
- *User-defined Attributes (UDA).* A relational dataset compression system must be able to admit new types of attributes that do not fit into either string, numerical, or categorical types.

In contrast to prior work [3, 6, 8, 12]—see Table 1—our system, Squish, can capture all of these desiderata. To support UDA (User Defined Attributes), Squish surfaces a new class interface, called the SquID (short for Squish Interface for Data types): users can instantiate new data types by simply implementing the required five functions. This interface is remarkably powerful, especially for datasets in specialized domains. For example, a new data type corresponding to genome sequence data can be implemented by a user using a few hundred lines of code. By encoding domain knowledge into the data type definition, we can achieve a significant higher compression rate than using "universal" compression algorithms like Lempel-Ziv [21].

The rest of this paper is organized as follows. In Section 2, we formally define the problem of relational dataset compression, and briefly explain the concepts of Arithmetic Coding. In Section 3 we discuss Bayesian Network learning and related issues, while details about Arithmetic Coding are discussed in Section 4. In Section 5, we prove the asymptotic optimality of the compression algorithm. In Section 6, we conduct experiments to compare Squish with prior systems and evaluate its running time and parameter sensitivity. We describe related work in Section 7. All our proofs can be found in our technical report [7], along with a brief review of Bayesian Networks and illustrative examples of our compression algorithm.

The source code of Squish is available on GitHub: https://github.com/Preparation-Publication-BD2K/db_compress

2. PRELIMINARIES

In this section, we define our problem more formally, and provide some background on Arithmetic Coding.

2.1 Problem Definition

We follow the same problem definition proposed by Babu et al. [3]. Suppose our dataset consists of a single relational table T, with n rows and m columns (our techniques extend to multi-relational case as well). Each row of the table is referred to as a tuple and each column of the table is referred to as an attribute. We assume that each attribute has an associated domain that is known to us. For instance, this information could be described when the table schema is specified.

The goal is to design a compression algorithm A and a decompression algorithm B, such that A takes T as input and outputs a compressed file $C(T)$, and B takes the compressed file

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

$C(T)$ as input and outputs T' as the approximate reconstruction of T. The goal is to minimize the file size of $C(T)$ while ensuring that the recovered T' is close enough to T.

The closeness constraint of T' to T is defined as follows: For each numerical attribute i, for each tuple t and the recovered tuple t', $|t_i - t'_i| \leq \varepsilon_i$, where t_i and t'_i are the values of attribute i of tuple t and t' respectively, and ε_i is error threshold parameter provided by the user. For non-numerical attributes, the recovered attribute value must be exactly the same as the original one: $t_i = t'_i$. Note that this definition subsumes lossless compression as a special case with $\varepsilon_i = 0$.

2.2 Arithmetic Coding

Arithmetic coding [19, 11] is a state-of-the-art adaptive compression algorithm for a sequence of dependent characters. Arithmetic coding assumes as a given a conditional probability distribution model for any character, conditioned on all preceding characters. If the sequence of characters are indeed generated from the probabilistic model, then arithmetic coding can achieve a near-entropy compression rate [11].

Formally, arithmetic coding is defined by a finite ordered alphabet $\mathscr{A}$, and a probabilistic model for a sequence of characters that specifies the probability distribution of each character X_k conditioned on all precedent characters $X_1, \ldots, X_{k-1}$. Let $\{a_n\}$ be any string of length n. To compute the encoded string for $\{a_n\}$, we first compute a probability interval for each character a_k:

$$[l_k, r_k] = [p(X_k < a_k | X_1 = a_1, \ldots, X_{k-1} = a_{k-1}), p(X_k \leq a_k | X_1 = a_1, \ldots, X_{k-1} = a_{k-1})]$$

We define the product of two probability interval as:

$$[l_1, r_1] \circ [l_2, r_2] = [l_1 + (r_1 - l_1) l_2, l_1 + (r_1 - l_1) r_2]$$

The probability interval for string $\{a_n\}$ is the product of probability intervals of all the characters in the string:

$$[l, r] = [l_1, r_1] \circ [l_2, r_2] \circ \ldots \circ [l_n, r_n]$$

Let k be the smallest integer such that there exists a non-negative integer $0 \leq M < 2^k$ satisfying:

$$l \leq 2^{-k} M, r \geq 2^{-k} (M+1)$$

Then the k-bit binary representation of M is the encoded bit string of $\{a_n\}$.

An example to illustrate how arithmetic coding works can be found in Figure 1. The three tables at the right hand side specify the probability distribution of the string $a_1 a_2 a_3$. The blocks at the left hand show the associated probability intervals for the strings: for example,

"aba" corresponds to $[0.12, 0.204] = [0, 0.4] \circ [0.3, 1] \circ [0, 0.3]$. As we can see, the intuition of arithmetic aoding is to map each possible string $\{a_n\}$ to disjoint probability intervals. By using these probability intervals to construct encoded strings, we can make sure that no code word is a prefix of another code word.

Notice that the length of the product of probability intervals is exactly the product of their lengths. Therefore, the length of each probability interval is exactly the same as the probability of the corresponding string $\mathbf{Pr}(\{a_n\})$. Using this result, the length of encoded string can be bounded as follows:

$$len(binary_\ code(\{a_n\})) \leq \lfloor -\log_2 \mathbf{Pr}(\{a_n\}) \rfloor + 2$$

3. STRUCTURE LEARNING

The overall workflow of Squish is illustrated in Figure 2. Squish uses a combination of Bayesian networks and arithmetic coding for compression. The workflow of the compression algorithm is the following:

1. Learn a Bayesian network structure from the dataset, which captures the dependencies between attributes in the structure graph, and models the conditional probability distribution of each attribute conditioned on all the parent attributes.
2. Apply arithmetic coding to compress the dataset, using the Bayesian network as probabilistic models.
3. Concatenate the model description file (describing the Bayesian network model) and compressed dataset file.

In this section, we focus on the first step of this workflow. We focus on the remaining steps (along with decompression) in Section 4.

Although the problem of Bayesian network learning has been extensively studied in literature [9], conventional objectives like Bayesian Information Criterion (BIC) [15] are suboptimal for the purpose of compressing datasets. In Section 3.1, we derive the correct objective function for learning a Bayesian network that minimizes the size of the compressed dataset and explain how to modify existing Bayesian network learning algorithms to optimize this objective function.

The general idea about how to apply arithmetic coding on a Bayesian network is straightforward: since the graph encoding the structure of a Bayesian Network is acyclic, we can use any topological order of attributes and treat the attribute values as the sequence of symbols in arithmetic coding. However, arithmetic coding does not naturally apply to non-categorical attributes. In Section 3.2, we introduce SquID, the mechanism for supporting non-categorical and arbitrary user-defined attribute types in Squish. SquID is the interface for every attribute type in Squish, and example SquIDs for categorical, numerical and string attributes are demonstrated in Section 3.3 to illustrate the wide applicability of this interface. We describe the SquID API in Section 3.4.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript

3.1 Learning a Bayesian Network: The Basics

Many Bayesian network learning algorithms search for the optimal Bayesian network by minimizing some objective function (e.g., negative log-likelihood, BIC [15]). These algorithms usually have two separate components [9]:

- A combinatorial optimization component that searches for a graph with optimal structure.
- A score evaluation component that evaluates the objective function given a graph structure.

The two components above are independent in many algorithms. In that case, we can modify an existing Bayesian network learning algorithm by changing the score evaluation component, while still using the same combinatorial optimization component. In other words, for any objective function, as long as we can efficiently evaluate it based on a fixed graph structure, we can modify existing Bayesian network learning algorithms to optimize it.

In this section, we derive a new objective function for learning a Bayesian network that minimizes the size of compressed dataset. We show that the new objective function can be evaluated efficiently given the structure graph. Therefore existing Bayesian Network learning algorithms can be used to optimize it.

Suppose our dataset D consists of n tuples, and each tuple t_i contains m attributes a_{i1}, $a_{i2}, \ldots, a_{im}$. Let $\mathscr{B}$ be a Bayesian network that describes a joint probability distribution over the attributes. Clearly, $\mathscr{B}$ contains m nodes, each corresponding to an attribute.

The total description length of D using $\mathscr{B}$ is $\mathbf{S}(D|\mathscr{B}) = \mathbf{S}(\mathscr{B}) + \mathbf{S}(Tuples|\mathscr{B})$, where $\mathbf{S}(\mathscr{B})$ is the size of description file of $\mathscr{B}$, and $\mathbf{S}(Tuples|\mathscr{B})$ is the total length of encoded binary strings of tuples using arithmetic coding. For the model description length $\mathbf{S}(\mathscr{B})$, we have $\mathbf{S}(\mathscr{B})=\sum_{i=1}^{m}\mathbf{S}(\mathscr{M}_i)$, where m is the number of attributes in our dataset, and $\mathscr{M}_1, \ldots, \mathscr{M}_m$ are the models for each attribute in $\mathscr{B}$. The expression $\mathbf{S}(Tuples|\mathscr{B})$ is just the sum of the $\mathbf{S}(t_i|\mathscr{B})$s (the lengthes of the encoded binary string for each t_i).

We have the following decomposition of $\mathbf{S}(t_i|\mathscr{B})$:

$$\mathbf{S}(t_i|\mathscr{B}) \approx -\sum_{j=1}^{m}\log_2\mathbf{Pr}(a_{ij}|parent(a_{ij}),\mathscr{M}_j)-\sum_{j=1}^{m}num(a_{ij})\log_2\varepsilon_j+\text{const}$$

where $parent(a_{ij})$ is the set of parent attributes of a_{ij} in $\mathscr{B}$, $num(a)$ is the indicator function of whether a is a numerical attribute or not, and ε_j is the maximum tolerable error for attribute a_{ij}. We will justify this decomposition in Section 3.3, after we introduce the encoding scheme for numerical attributes.

Therefore, the total description length $\mathbf{S}(D|\mathscr{B})$ can be decomposed as follows:

$$\mathbf{S}(D|\mathscr{B}) \approx \sum_{j=1}^{m}[\mathbf{S}(\mathscr{M}_j) - \sum_{t_i \in DB} \log_2 \mathbf{Pr}(a_{ij}|parent(a_{ij}), \mathscr{M}_j)] - n(\sum_{j=1}^{m} num(a_j)\log_2 \varepsilon_j + \text{const})$$

Note that the term in the second line does not depend on either $\mathscr{B}$ or $\mathscr{M}_i$. Therefore we only need to optimize the first summation. We denote each term in the first summation on the right hand side as obj_j:

$$obj_j = \mathbf{S}(\mathscr{M}_j) - \sum_{t_i \in DB} \log \mathbf{Pr}(a_{ij}|parent(a_{ij}), \mathscr{M}_j)$$

For each obj_j, if the network structure (i.e., $parent(a_{ij})$) is fixed, then obj_j only depends on $\mathscr{M}_j$. In that case, optimizing $\mathbf{S}(D|\mathscr{B})$ is equivalent to optimizing each obj_j individually. In other words, if we fix the Bayesian network structure in advance, then the parameters of each model can be learned separately.

Optimizing obj_j on $\mathscr{M}_j$ is exactly the same as maximizing like-lihood. For many models, a closed-form solution for identifying maximum likelihood parameters exists. In such cases, the optimal $\mathscr{M}_j$ can be quickly determined and the objective function $\mathbf{S}(D|\mathscr{B})$ can be computed efficiently.

Structure Learning—In general, searching for the optimal Bayesian network structure is NP-hard [9]. In Squish, we implemented a simple greedy algorithm for this task. The algorithm starts with an empty seed set, and repeatedly finds new attributes with the lowest obj_j, and adds these new attributes to the seed set. The pseudo-code can be found in our technical report [7].

The greedy algorithm has a worst case time complexity of $O(m^4 n)$ where m is the number of columns and n is the number of tuples in the dataset. For large datasets, even this simple greedy algorithm is not fast enough. However, note that the objective values obj_j are only used to compare different models. So we do not require exact values for them, and some rough estimation would be sufficient. Therefore, we can use only a subset of data for the structure learning to improve efficiency.

3.2 Supporting Complex Attributes

Encoding and Decoding Complex Attributes—Before applying arithmetic coding on a Bayesian network to compress the dataset as we stated earlier, there are two issues that we need to address first:

- Arithmetic Coding requires a finite alphabet for each symbol. However, it is natural for attributes in a dataset to have infinite range (e.g., numerical attributes, strings).
- In order to support user-defined data types, we need to allow the user to specify a probability distribution over an unknown data type.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

To address these difficulties, we introduce the concept of SquID, short for Squish Interface for Data types. A SquID is a (possibly infinite) decision tree [18] with non-negative probabilities associated with edges in the tree, such that for every node v, the probabilities of all the edges connecting v and v's children sum to one.

Figure 3 shows an example infinite SquID for a positive numerical attribute X. As we can see, each edge is associated with a decision rule and a non-negative probability. For each non-leaf node v_{2k-1}, the decision rules on the edges between v_{2k-1} and its children v_{2k} and v_{2k+1} are $x \leq k$ and $x > k$ respectively. Note that these two decision rules do not overlap with each other and covers all the possibilities. The probabilities associated with these two rules sum to 1, which is required in SquID. This SquID describes the following probability distribution over X:

$$\mathbf{Pr}(X \in (k-1, k]) = 0.9^{k-1} \times 0.1$$

In Section 4, we will show that we can encode or decode an attribute using Arithmetic Coding if the probability distribution of this attribute can be represented by a SquID.

As shown in Figure 3, a SquID naturally controls the maximum tolerable error in a lossy compression setting. Each leaf node v corresponds to a subset A_v of attribute values such that for every $a \in A_v$, if we start from the root and traverse down according to the decision rules, we will eventually reach v. As an example, in Figure 3, for each leaf node v_{2k} we have $A_{v_{2k}} = (k - 1, k]$. Let a_v be the representative attribute value of a leaf node v, then the maximum possible recovery error for v is:

$$\varepsilon_v = \sup_{a \in A_v} \text{distance}(a, a_v)$$

Let T_i be the SquID corresponding to the ith attribute. As long as for every $v \in T_i$, ε_v is less than or equal to the maximum tolerable error ε_i, we can satisfy the closeness constraint (defined in Section 2.1).

Using User-defined Attributes as Predictors—To allow user-defined attributes to be used as predictors for other attributes, we introduce the concept of attribute interpreters, which translate attributes into either categorical or numerical values. In this way, these attributes can be used as predictors for other attributes.

The attribute interpreters can also be used to capture the essential features of an attribute. For example, a numerical attribute could have a categorical interpreter that better captures the internal meaning of the attribute. This process is similar to the feature extraction procedure in many data mining applications, and may improve compression rate.

3.3 Example SquIDs

In Squish, we have implemented models for three primitive data types. We intended these models to both illustrate how SquIDs can be used to define probability distributions, and

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

also to cover most of the basic data types, so the system can be used directly by casual users without writing any code.

We implemented models for the following types of attributes:

- Categorical attributes with finite range.
- Numerical attributes, either integer or float number.
- String attributes

Categorical Attributes—The distribution over a categorical attributes can be represented using a trivial one-depth SquID.

Numerical Attributes—For a numerical attribute, we construct the SquID using the idea of *bisection*. Each node v is marked with an upper bound v_r and a lower bound v_l, so that every attribute value in range $(v_l, v_r]$ will pass by v on its path from the root to the corresponding leaf node. Each node has two children and a bisecting point v_m, such that the two children have ranges $(v_l, v_m]$ and $(v_m, v_r]$ respectively. The branching process stops when the range of the interval is less than 2ε, where ε is the maximum tolerable error. Figure 4 shows an example SquID for numerical attributes.

Since each node represents a continuous interval, we can compute its probability using the cumulative distribution function. The branching probability of each node is:

$$(\mathbf{Pr}(\textit{left branch}), \mathbf{Pr}(\textit{right branch})) = \left(\frac{\mathbf{Pr}(v_l < X \le v_m)}{\mathbf{Pr}(v_l < X \le v_r)}, \frac{\mathbf{Pr}(v_m < X \le v_r)}{\mathbf{Pr}(v_l < X \le v_r)}\right)$$

Clearly, the average number of bits that is needed to encode a numerical attribute depends on both the probability distribution of the attribute and the maximum tolerable error ε. The following theorem gives us a lower bound on the average number of bits that is necessary for encoding a numerical attribute (the proof can be found in our technical report [7]):

Theorem 1: *Let $X \in \mathscr{X} \subseteq \mathbb{R}$ be a numerical random variable with continuous support $\mathscr{X}$ and probability density function $f(X)$. Let $g : \mathscr{X} \to \{0, 1\}^*$ be any uniquely decodable encoding function, and $h : \{0, 1\}^* \to \mathscr{X}$ be any decoding function. If there exists a function $\rho : \mathscr{X} \to \mathbb{R}^+$ such that:*

$$\forall x, y \in \mathscr{X}, |x - y| < 2\varepsilon \Rightarrow |f(x) - f(y)| \le \rho(x) f(x) |x - y| \quad (1)$$

and g, h satisfies the ε-closeness constraint:

$$\forall x \in \mathscr{X}, |h(g(x)) - x| \le \varepsilon$$

Then

$$E_x[len(g(X))] \geq E_x[-\log_2 f(X)] - E_x[\log_2(2\varepsilon\rho(X)+1)] - \log_2\varepsilon - 2$$

Furthermore, if g is the bisecting code described above, then

$$E_x[len(g(X))] \leq E_x[-\log_2 f(X)] - \log_2 l + E_x[\max(\log_2\rho(X)+\log_2 l, 0)] + 4$$

where $l = \min_v(v_r - v_l)$ *is the minimum length of probability intervals in the tree.*

Equation (1) is a mild assumption that holds for many common probability distributions, including uniform distribution, Gaussian distribution, and Laplace distribution [2].

To understand the intuition behind the results in Theorem 1. Let us consider a Gaussian distribution as an example:

$$f(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

In this case,

$$\rho(x) = \frac{|x-\mu|+2\varepsilon}{\sigma^2}$$

Substituting into the first expression, we have:

$$E_X[len(g(X))] \geq \log_2\sigma - \log_2\varepsilon + \log_2\sqrt{2\pi} - \frac{3}{2} - E_X[\log_2(\frac{2\varepsilon(|X-\mu|+2\varepsilon)}{\sigma^2}+1)]$$

Note that when ε is small compared to σ (e.g., $\varepsilon < \frac{1}{10}\sigma$), the last term is approximately zero. Therefore, the number of bits needed to compress X is approximately $\log_2\frac{\sigma}{\varepsilon}$.

Now consider the second result,

$$E_X[len(g(X))] \leq \log_2\sigma - \log_2\frac{l}{2} + \log_2\sqrt{2\pi} + \frac{7}{2} + E_X[\max(\log_2\frac{l(|X-\mu|+2\varepsilon)}{\sigma^2}, 0)]$$

Let $l = 2\varepsilon$, and when $\varepsilon < \frac{1}{10}\sigma$, the last term is approximately zero. Comparing the two results, we can see that the bisecting scheme achieves near optimal compression rate.

Theorem 1 can be used to justify the decomposition in Section 3.1. Recall that we used the following expression as an approximation of $len(g(X))$:

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

$$len(g(X)) \approx -\log_2 f(X) - \log_2 \varepsilon + \text{const}$$

Compared to either the upper bound or lower bound in Theorem 1, the only term we omitted is the term related to $\rho(X)$. As we have seen in the Gaussian case, this term is approximately zero when ε is smaller than $\frac{\sigma}{10}$ where σ is the standard deviation parameter. The same conclusion is also true for the Laplace distribution [2] and the uniform distribution.

String Attributes—The SquID for string attributes can be viewed as having two steps:

1. determine the length of the string
2. determine the characters of the string

The length of a string can be viewed as an integer, so we can use exactly the same bisecting rules as for numerical attributes. After that, we use n more steps to determine each character of the string, where n is the string's length. The probability distribution of each character can be specified by conventional probabilistic models like the k-gram model.

3.4 SquID API

In Squish, SquID is defined as an abstract class [1]. There are five functions that are required to be implemented in order to define a new data type using SquID. These five functions allow the system to interactively explore the SquID class: initially, the current node pointer is set to the root of SquID; each function will either acquire information about the current node, or move the current node pointer to one of its children. Table 2 lists the five functions together with their high level description.

We also develop another abstract class called SquIDModel. A SquIDModel first reads in all the tuples in the dataset, then generates a SquID instance and an estimation of the objective value obj_j derived in Section 3.1:

$$obj_j = \mathbf{S}(\mathscr{M}_j) - \sum_{t_i \in DB} \log \mathbf{Pr}(a_{ij} | parent(a_{ij}), \mathscr{M}_j)$$

There are two reasons behind this design:

- For a parametric SquID, the parameters need to be learned using the dataset at hand.
- The Bayesian network learning algorithm requires an estimation of the objective value. Although it is possible to compute the objective value by actually compressing the attributes, in many cases it is much more efficient to approximate it directly.

SquIDModel requires six functions to be implemented. These functions allow the SquIDModel to iterate over the dataset and generate SquID instances. The specification of these functions and the psuedo-code of their interactions with SquID can be found in our technical report [7].

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

A SquIDModel instance is initialized with the target attribute and the set of predictor attributes. After that, the model instance will read over all tuples in the dataset and need to decide the optimal choice of parameters. The model also needs to return an estimate of the objective value obj_j, which will be used in the Bayesian network structure learning algorithm to compare models. Finally, SquIDModel should be able to generate SquID instances based on parent attribute values.

4. COMPRESSION AND DECOMPRESSION

In this section, we discuss how we can use arithmetic coding correctly for compression and decompression given a Bayesian Network. In Section 4.1, we discuss implementation details that ensure the correctness of arithmetic coding using finite precision float numbers. In Section 4.2 we describe the decompression algorithm.

4.1 Compression

We use the same notation as in Section 3.1: a tuple t contains m attributes, and without loss of generality we assume that they follow the topological order of the Bayesian network:

$$t=\{a_1,\ldots,a_m\}, parent(a_j) \subseteq \{a_1,\ldots,a_{j-1}\}$$

We first compute a probability interval for each branch in a SquID. For each SquID T, we define PI_T as a mapping from branches of T to probability intervals. The definition is similar to the one in Section 2.2: let v be any non-leaf node in T, suppose v has k children $u_1, \ldots, u_k$, and the edge between v and u_i is associated with probability p_i, then $\mathrm{PI}_T(v \to u_i)$ is defined as:

$$\mathrm{PI}(v \to u_i)=[\sum_{j<i} p_j, \sum_{j\le i} p_j]$$

Now we can compute the probability interval of t. Let T_j be the SquID for a_j conditioned on its parent attributes. Denote the leaf node in T_j that a_j corresponds to as v_j. Suppose the path from the root of T_j to v_j is $u_{j1} \to u_{j2} \to \ldots \to u_{jk_j} \to v_j$. Then, the probability interval of tuple t is:

$$\begin{aligned}[L,R]=&\mathrm{PI}_{T_1}(u_{11} \to u_{12}) \circ \ldots \circ \mathrm{PI}_{T_1}(u_{1k_1} \to v_1) \circ \\ &\mathrm{PI}_{T_2}(u_{21} \to u_{22}) \circ \ldots \circ \mathrm{PI}_{T_2}(u_{2k_2} \to v_2) \circ \ldots \circ \\ &\mathrm{PI}_{T_m}(u_{m1} \to u_{m2}) \circ \ldots \circ \mathrm{PI}_{T_m}(u_{mk_m} \to v_m)\end{aligned}$$

where $\circ$ is the probability interval multiplication operator defined in Section 2.2. The code string of tuple **t** corresponds to the largest subinterval of $[L, R]$ of the form $[2^{-k}M, 2^{-k}(M+1)]$ as described in Section 2.2.

In practice, we cannot directly compute the final probability interval of a tuple: there could be hundreds of probability intervals in the product, so the result can easily exceed the precision limit of a floating-point number.

Author Manuscript

Algorithm 1 shows the pseudo-code of the precision-aware compression algorithm. We leverage two tricks to deal with the finite precision problem: the classic early bits emission trick [10] is described in Section 4.1.1; the new deterministic approximation trick is described in Section 4.1.2.

Algorithm 1

Encoding Algorithm

```
function ArithmeticCoding([l1, r1], … , [ln, rn])
  code ← θ
  It ← [0, 1]
  for i = 1to n do
    It ← It ∘ [li, ri]
    while ∃k = 0or 1, It ⊆ [k/2, (k+1)/2] do
      code ← code + k
      It ← [2It.l − k, 2It.r − k]
    end while
  end for
  Find smallest k such that
      ∃M, [2^−k M, 2^−k (M + 1)] ⊆ It
  return code +M
end function
```

4.1.1 Early Bits Emission—Without loss of generality, suppose

$$[L, R] = [l_1, r_1] \circ [l_2, r_2] \circ \ldots \circ [l_n, r_n]$$

Define $[L_i, R_i]$ as the product of first i probability intervals:

$$[L_i, R_i] = [l_1, r_1] \circ [l_2, r_2] \circ \ldots \circ [l_i, r_i]$$

If there exist positive integer k_i and non-negative integer M_i such that

$$2^{-k_i} M_i \le L_i < R_i \le 2^{-k_i}(M_i + 1)$$

Then the first k_i bits of the code string of t must be the binary representation of M_i. Define

$$[L_i', R_i'] = [2^{k_i} L_i - M_i, 2^{k_i} R_i - M_i]$$

Then it can be verified that

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

$$binary_code([L,R])=binary_code(M_i)+binary_code([L_i^{'},R_i^{'}]\circ[l_{i+1},r_{i+1}]\ldots\circ[l_n,r_n])$$

Therefore, we can immediately output the first k_i bits of the code string. After that, we compute the product:

$$[L_i^{'},R_i^{'}]\circ[l_{i+1},r_{i+1}]\ldots\circ[l_n,r_n]$$

We can recursively use the same early bit emitting scheme for this product. In this way, we can greatly reduce the likelihood of precision overflow.

4.1.2 Deterministic Approximation—For probability intervals containing 0.5, we cannot emit any bits early. In rare cases, such a probability interval would exceed the precision limit, and the correctness of our algorithm would be compromised.

To address this problem, we introduce the deterministic approximation trick. Recall that the correctness of arithmetic coding relies on the non-overlapping property of the probability intervals. Therefore, we do not need to compute probability intervals with perfect accuracy: the correctness is guaranteed as long as we ensure these probability intervals do not overlap with each other.

Formally, let t_1, t_2 be two different tuples, and suppose their probability intervals are:

$$\begin{aligned}\mathrm{PI}(t_1)&=[l_1,r_1]=[l_{11},r_{11}]\circ[l_{12},r_{12}]\circ\ldots\circ[l_{1n_1},r_{1n_1}]\\ \mathrm{PI}(t_2)&=[l_2,r_2]=[l_{21},r_{21}]\circ[l_{22},r_{22}]\circ\ldots\circ[l_{2n_2},r_{2n_2}]\end{aligned}$$

The deterministic approximation trick is to replace $\circ$ operator with a deterministic operator $\diamond$ that approximates $\circ$ and has the following properties:

- For any two probability intervals [a, b] and [c, d]:

$$[a,b]\diamond[c,d]\subseteq[a,b]\circ[c,d]$$

- For any two probability intervals [a, b] and [c, d] with $b-a\geq\epsilon$ and $d-c\geq\epsilon$. Let [l, r] = [a, b] $\diamond$ [c, d], then:

$$\exists k,M,2^{-k}M\leq l<r\leq 2^{-k}(M+1),2^k(r-l)\geq\varepsilon$$

In other words, the product computed by $\diamond$ operator is always a subset of the product computed by $\circ$ operator, and $\diamond$ operator always ensures that the product probability interval has length greater than or equal to ϵ after emitting bits. The first property guarantees the non-overlapping property still holds, and the second property prevents potential precision overflow. As we will see in Section 4.2, these two properties are sufficient to guarantee the correctness of arithmetic coding.

4.2 Decompression

When decompressing, Squish first reads in the dataset schema and all of the model information, and stores them in the main memory. After that, it scans over the compressed dataset, extracts and decodes the binary code strings to recover the original tuples.

Algorithm 2

Decoding Algorithm

---

function Decoder.Initialization

  $I_b \leftarrow [0, 1]$

  $I_t \leftarrow [0, 1]$

end function

function Decoder.GetNextBranch(*branches*)

  while not $\exists br \in branches, I_b \subseteq I_t \diamond \mathrm{PI}(br)$ **do**

    Read in the next bit x

    $I_b \leftarrow I_b \circ \left[\frac{x}{2}, \frac{x+1}{2}\right]$

  end while

  if $I_b \subseteq I_t \diamond \mathrm{PI}(br), br \in branches$ **then**

    $I_t \leftarrow I_t \diamond \mathrm{PI}(br)$

    while $\exists k = 0 \text{or } 1, I_t \subseteq \left[\frac{k}{2}, \frac{k+1}{2}\right]$ **do**

      $I_t \leftarrow [2I_t.l - k, 2I_t.r - k]$

      $I_b \leftarrow [2I_b.r - k, 2I_b.r - k]$

    end while

    return *br*

  end if

end function

Algorithm 2 describes the procedure to decide the next branch. The decoder maintains two probability intervals I_b and I_t. I_b is the probability interval corresponding to all the bits that the algorithm has read in so far. I_t is the probability interval corresponding to all decoded attributes. At each step, the algorithm computes the product of I_t and the probability interval for every possible attribute value, and then checks whether I_b is contained by one of those probability intervals. If so, we can decide the next branch, and update I_t accordingly. If not, we continue reading in the next bit and update I_b.

By calling Algorithm 2 repeatedly, we can gradually decode the whole tuple. The full decoding procedure with an illustrative example can be found in our technical report [7].

Notice that Algorithm 2 mirrors Algorithm 1 in the way it computes probability interval products. This design is to ensure that the encoding and decoding algorithm always apply the same deterministic approximation that we described in Section 4.1.2. The following theorem states the correctness of the algorithm (the proof can be found in our technical report [7]):

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Theorem 2: *Let* $[l_1, r_1], \dots, [l_n, r_n]$ *be probability intervals with* $r_i - l_i \geq \epsilon$ *where* ϵ *is the small constant defined in Section 4.1.2. Let s be the output of* Algorithm 1 *on these probability intervals. Then* Algorithm 2 *can always determine the correct branch from alternatives using s as input:*

$$PI(Decoder.GetNextBranch(branch_i)) = [l_i, r_i]$$

5. DISCUSSION: OPTIMALITY

We can prove that Squish achieves asymptotic near-optimal compression rate for lossless compression if the dataset only contains categorical attributes and can be described efficiently using a Bayesian network (the proof can be found in [7]):

Theorem 3: *Let* $a_1, a_2, \dots, a_m$ *be categorical attributes with joint probability distribution* $P(a_1, \dots, a_m)$ *that decomposes as*

$$P(a_1, \dots, a_m) = \prod_{i=1}^{m} P(a_i | parent_i)$$

such that

$$parent_i \subseteq \{a_1, \dots, a_{i-1}\}, card(parent_i) \leq c$$

Suppose the dataset $\mathscr{D}$ *contains n tuples that are i.i.d. samples from P. Let* $M = \max_i card(a_i)$ *be the maximum cardinality of attribute range. Then* Squish *can compress* $\mathscr{D}$ *using less than* $H(\mathscr{D}) + 4n + 32mM^{c+1}$ *bits on average, where* $H(\mathscr{D})$ *is the entropy* [5] *of the dataset* $\mathscr{D}$.

Thus, when n is large, the difference between the size of the compressed dataset using our system and the entropy[1] of $\mathscr{D}$ is at most $5n$, that is only 5 bits per tuple. This indicates that Squish is asymptotically near-optimal for this setting.

When the dataset $\mathscr{D}$ contains numerical attributes, the entropy $H(\mathscr{D})$ is not defined, and the techniques we used to prove Theorem 3 no longer apply. However, in light of Theorem 1, it is likely that Squish still achieves asymptotic near-optimal compression.

6. EXPERIMENTS

In this section, we evaluate the performance of Squish against the state of the art semantic compression algorithms SPARTAN [3] and ItCompress [8] (see Table 1). For reference we also include the performance of gzip [21], a well-known syntactic compression algorithm.

[1] By Shannon's source coding theorem [5], there is no algorithm that can achieve compression rate higher than entropy.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

We use the following four publicly available datasets:

- *Corel* (http://kdd.ics.uci.edu/databases/CorelFeatures) is a 20 MB dataset containing 68,040 tuples with 32 numerical color histogram attributes.
- *Forest-Cover* (http://kdd.ics.uci.edu/databases/covertype) is a 75 MB dataset containing 581,000 tuples with 10 numerical and 44 categorical attributes.
- *Census* (http://thedataweb.rm.census.gov/ftp/cps_ftp.html) is a 610MB dataset containing 676,000 tuples with 36 numerical and 332 categorical attributes.
- *Genomes* (ftp://ftp.1000genomes.ebi.ac.uk) is a 18.2GB dataset containing 1,832,506 tuples with about 10 numerical and 2500 categorical attributes.[2]

The first three datasets have been used in previous papers [3, 8], and the compression ratio achieved by SPARTAN, ItCompress and gzip on these datasets have been reported in Jagadish et al.'s work [8]. We did not reproduce these numbers and only used their reported performance numbers for comparison. For the *Census* dataset, the previous papers only used a subset of the attributes in the experiments (7 categorical attributes and 7 numerical attributes). Since we are unaware of the selection criteria of the attributes, we are unable to compare with their algorithms, and we will only report the comparison with gzip.

For the *Corel* and *Forest-Cover* datasets, we set the error tolerance as a percentage (1% by default) of the width of the range for numerical attributes as in previous work. For the *Census* dataset, we set all error tolerance to 0 (i.e. the compression is lossless). For the *Genomes* dataset, we set the error tolerance for integer attributes to 0 and float number attributes to 10^{-8}.

In all experiments, we only used the first 2000 tuples in the structure learning algorithm to improve efficiency. All available tuples are used in other parts of the algorithm.

6.1 Compression Rate Comparison

Figure 5 shows the comparison of compression rate on the *Corel* and *Forest-Cover* datasets. In these figures, X axis is the error tolerance for numerical attributes (% of the width of range), and Y axis is the compression ratio, defined as follows:

$$\text{compression ratio} = \frac{\text{data size with compression}}{\text{data size without compression}}$$

As we can see from the figures, Squish significantly outperforms the other algorithms. When the error tolerance threshold is small (0.5%), Squish achieves about *50% reduction in compression ratio on the Forest Cover dataset and 75% reduction on the Corel dataset, compared to the nearest competitor ItCompress (gzip)*, which applies gzip algorithm on top of the result of ItCompress. The benefit of not using gzip as a post-processing step is that we can still permit tuple-level access without decompressing a larger unit.

[2]In this dataset, many attributes are optional and these numbers indicate the average number of attributes that appear in each tuple.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

The remarkable superiority of our system in the Corel dataset reflects the advantage of Squish in compressing numerical attributes. Numerical attribute compression is known to be a hard problem [14] and none of the previous systems have effectively addressed it. In contrast, our encoding scheme can leverage the skewness of the distribution and achieve near-optimal performance.

Figure 6 shows the comparison of compression rate on the *Census* and *Genomes* datasets. Note that in these two datasets, we set the error tolerance threshold to be extremely small, so that the compression is essentially lossless. As we can see, even in the lossless compression scenario, our algorithm still outperforms gzip significantly. *Compared to gzip,* Squish *achieves* 48% *reduction in compression ratio in Census dataset and* 56% *reduction in Genomes dataset.*

6.2 Compression Breakdown

As we have seen in the last section, Squish achieved superior compression ratio in all four datasets. In this section, we use detailed case studies to illustrate the reason behind the significant improvement over previous papers.

6.2.1 Categorical Attributes—In this section, we study the source of the compression in Squish for categorical attributes. We will use three different treatments for the categorical attributes and see how much compression is achieved for each of these treatments:

- Domain Code: We replace the categorical attribute values with short binary code strings. Each code string has length $\lceil \log_2 N \rceil$, where N is the total number of possible categorical values for the attribute.
- Column: We ignore the correlations between categorical attributes and treat all the categorical attributes as independent.
- Full: We use both the correlations between attributes and the skewness of attribute values in our compression algorithm.

We will use the *Genomes* and *Census* dataset here since they consist of mostly categorical attributes. We keep the compression algorithm for numerical attributes unchanged in all treatments. Figure 7 shows the compression ratio of the three treatments:

As we can see, the compression ratio of the basic domain coding scheme can be improved up to 70% if we take into account the skewness of the distribution in attribute values. Furthermore, the correlation between attributes is another opportunity for compression, which improved the compression ratio by 50% in both datasets.

An interesting observation is that the Column treatment achieves comparable compression ratio as gzip in both datasets, which suggests that gzip is in general capable of capturing the skewness of distribution for categorical attributes, but unable to capture the correlation between attributes.

6.2.2 Numerical Attributes—We now study the source of the compression in Squish for numerical attributes. We use the following five treatments for the numerical attributes:

- IEEE Float: We use the IEEE Single Precision Floating Point standard to store all attributes.
- Discrete: Since all attributes in the dataset have value between 0 and 1, we use integer i to represent a float number in range $[\frac{i}{10^7}, \frac{i+1}{10^7}]$, and then store each integer using its 24-bit binary representation.
- Column: We ignore the correlation between numerical attributes and treat all attributes as independent.
- Full: We use both the correlations between attributes and distribution information about attribute values.
- Lossy: The same as the Full treatment, but we set the error tolerance at 10^{-4} instead.

We use the *Corel* dataset here since it contains only numerical attributes. The error tolerance in all treatments except the last are set to be 10^{-7} to make sure the comparison is fair (IEEE single format has precision about 10^{-7}). All the numerical attributes in this dataset are in range [0, 1], with a distribution peaked at 0. Figure 8 shows the compression ratio of the five treatments.

As we can see, storing numerical attributes as float numbers instead of strings gives us about 55% compression. However, the compression rate can be improved by another 50% if we recognize distributional properties (i.e., range and skewness). Utilizing the correlation between attributes in the *Corel* dataset only slightly improved the compression ratio by 3%. Finally, we see that the benefit of lossy compression is significant: even though we only reduced the precision from 10^{-7} to 10^{-4}, the compression ratio has already been improved by 50%.

6.3 Running Time

Table 3 lists the running time of the five components in Squish. All experiments are performed on a computer with eight[3] 3.4GHz Intel Xeon processors. For the *Genomes* dataset, which contains 2500 attributes—an extremely large number—we constructed the Bayesian Network manually. Note that none of the previous papers have been applied on a dataset with the magnitude of the *Genomes* dataset (both in number of tuples and number of attributes).

As we can see from Table 3, our compression algorithm scales reasonably: even with the largest dataset *Genomes*, the compression can still be finished within hours. Recall that since our algorithm is designed for archival not online query processing, and *our goal is therefore to minimize storage as much as possible*, a few hours for large datasets is adequate.

The running time of the parameter tuning component can be greatly reduced if we use only a subset of tuples (as we did for structure learning). The only potential bottleneck is structure learning, which scales badly with respect to the number of attributes ($O(m^4)$). To handle

[3]The implementation is single-threaded, so only one processor is used.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

datasets of this scale, another approach is to partition the dataset column-wise, and apply the compression algorithm on each partition separately. We plan to investigate this in future work.

We remark that, unlike gzip [21], Squish allows random access of tuples without decompressing the whole dataset. Therefore, if users only need to access a few tuples in the dataset, then they will only need to decode those tuples, which would require far less time than decoding the whole dataset.

6.4 Sensitivity to Bayesian Network Learning

We now investigate the sensitivity of the performance of our algorithm with respect to the Bayesian network learning. We use the *Census* dataset here since the correlation between attributes in this dataset is stronger than other datasets, so the quality of the Bayesian network can be directly reflected in the compression ratio.

Since our structure learning algorithm only uses a subset of the training data, one might question whether the selection of tuples in the structure learning component would affect the compression ratio. To test this, we run the algorithm for five times, and randomly choose the tuples participating in the structure learning. Table 4 shows the compression ratio of the five runs. As we can see, the variation between runs are insignificant, suggesting that our compression algorithm is robust.

We also study the sensitivity of our algorithm with respect to the number of tuples used for structure learning. Table 5 shows the compression ratio when we use 1000, 2000 and 5000 tuples in the structure learning algorithm respectively. As we can see, the compression ratio improves gradually as we use more tuples for structure learning.

7. RELATED WORK

Although compression of datasets is a classical research topic in the database research community [14], the idea of exploiting attribute correlations (a.k.a. semantic compression) is relatively new. Babu et al. [3] used functional dependencies among attributes to avoid storing them explicitly. Jagadish et al. [8] used a clustering algorithm for tuples. Their compression scheme stores, for each cluster of tuples, a representative tuple and the differences between the representative tuple and other tuples in the cluster. These two types of dependencies are special cases of the more general Bayesian network style dependencies used in this paper.

The idea of applying arithmetic coding on Bayesian networks was first proposed by Davies and Moore [6]. However, their work only supports categorical attributes (a simple case). Further, the authors did not justify their approach by either theoretically or experimentally comparing their algorithm with other semantic compression algorithms. Lastly, they used conventional BIC [15] score for learning a Bayesian Network, which is suboptimal, and their technique does not apply to the lossy setting.

The compression algorithm developed by Raman and Swart [12] used Huffman Coding to compress attributes. Therefore, their work can only be applied to categorical attributes and

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

can not fully utilize attribute correlation (the authors only mentioned that they can exploit attribute correlations by encoding multiple attributes at once). The major contribution of Raman's work [12] is that they formalized the old idea of compressing ordered sequences by storing the difference between adjacent elements, which has been used in search engines to compress inverted indexes [4] and also in column-oriented database systems [16].

Bayesian networks are well-known general purpose probabilistic models to characterize dependencies between random variables. For reference, the textbook written by Koller and Friedman [9] covers many recent developments. Arithmetic coding was first introduced by Rissanen [13] and further developed by Witten et al. [19]. An introductory paper written by Langdon Jr. [10] covers most of the basic concepts of Arithmetic Coding, including the early bit emission trick. The deterministic approximation trick is original. Compared to the overflow prevention mechanism in Witten et al.'s work [19], the deterministic approximation trick is simpler and easier to implement.

8. CONCLUSION

In this paper, we propose Squish, an extensible system for compressing relational datasets. Squish exploits both correlations between attributes and skewness of numerical attributes, and thereby achieves better compression rates than prior work. We also develop SquID, an interface for supporting user-defined attributes in Squish. Users can use SquID to define new data types by simply implementing a handful of functions. We develop new encoding schemes for numerical attributes using SquID, and prove its optimality. We also discuss mechanisms for ensuring the correctness of Arithmetic Coding in finite precision systems. We prove the asymptotic optimality of Squish on any dataset that can be efficiently described using a Bayesian Network. Experiment results on two real datasets indicate that Squish significantly outperforms prior work, achieving more than 50% reduction in storage.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grant IIS-1513407 awarded by the National Science Foundation, grant 1U54GM114838 awarded by NIGMS and 3U54EB020406-02S1 awarded by NIBIB through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative (www.bd2k.nih.gov), the Siebel Energy Institute, and the Faculty Research Award provided by Google. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

References

1. C++ abstract class reference. http://en.cppreference.com/w/cpp/language/abstract_class
2. Abramowitz, M., Stegun, IA. Handbook of mathematical functions: with formulas, graphs, and mathematical tables. Vol. 55. Courier Corporation; 1964.
3. Babu, S., Garofalakis, M., Rastogi, R. ACM SIGMOD Record. Vol. 30. ACM; 2001. Spartan: A model-based semantic compression system for massive data tables; p. 283-294.
4. Baeza-Yates, R., Ribeiro-Neto, B., et al. Modern information retrieval. Vol. 463. ACM press; New York: 1999.
5. Cover, TM., Thomas, JA. Elements of information theory. John Wiley & Sons; 2006.
6. Davies, S., Moore, A. Bayesian networks for lossless dataset compression. Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining; ACM; 1999. p. 387-391.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

7. Gao, Y., Parameswaran, A. Squish: Near-optimal compression for archival of relational datasets. http://arxiv.org/abs/1602.04256
8. Jagadish, H., Ng, RT., Ooi, BC., Tung, A. Proceedings of the 20th International Conference on Data Engineering. IEEE; 2004. Itcompress: An iterative semantic compression algorithm; p. 646-657.
9. Koller, D., Friedman, N. Probabilistic graphical models: principles and techniques. MIT press; 2009.
10. Langdon GG Jr. An introduction to arithmetic coding. IBM Journal of Research and Development. 1984; 28(2):135–149.
11. MacKay, DJ. Information theory, inference, and learning algorithms. Vol. 7. Cambridge university press; 2003.
12. Raman, V., Swart, G. How to wring a table dry: Entropy compression of relations and querying of compressed relations. Proceedings of the 32nd international conference on Very large data bases; VLDB Endowment; 2006. p. 858-869.
13. Rissanen J. Generalized kraft inequality and arithmetic coding. IBM Journal of research and development. 1976; 20(3):198–203.
14. Roth MA, Van Horn SJ. Database compression. ACM Sigmod Record. 1993; 22(3):31–39.
15. Schwarz G, et al. Estimating the dimension of a model. The annals of statistics. 1978; 6(2):461–464.
16. Stonebraker, M., Abadi, DJ., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. C-store: a column-oriented dbms. Proceedings of the 31st international conference on Very large data bases; VLDB Endowment; 2005. p. 553-564.
17. Willems FM, Shtarkov YM, Tjalkens TJ. The context-tree weighting method: basic properties. Information Theory, IEEE Transactions on. 1995; 41(3):653–664.
18. Witten, IH., Frank, E. Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann; 2005.
19. Witten IH, Neal RM, Cleary JG. Arithmetic coding for data compression. Communications of the ACM. 1987; 30(6):520–540.
20. Wu, K., Shoshani, A., Otoo, E. Word aligned bitmap compression method, data structure, and apparatus. US Patent. 6,831,575.. Dec 14. 2004
21. Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE Transactions on information theory. 1977; 23(3):337–343.

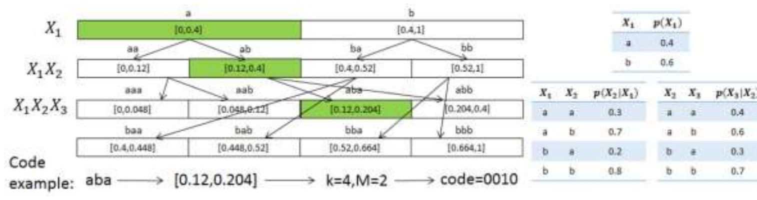


Figure 1.
Arithmetic Coding Example

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

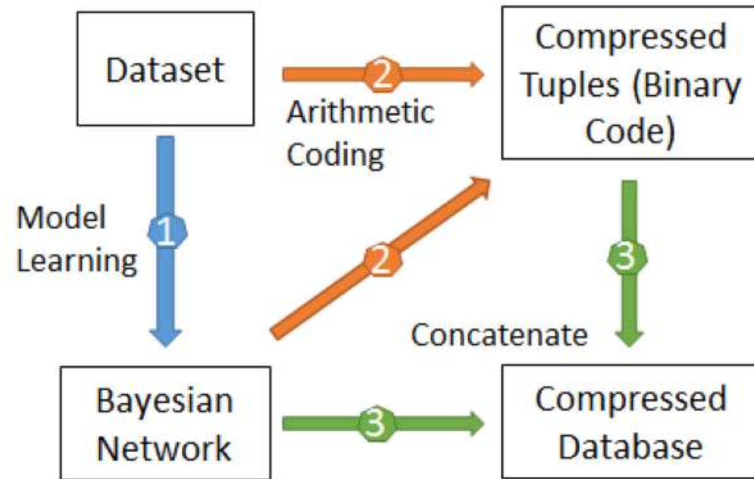


Figure 2.
Workflow of the Compression and Decompression Algorithm

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

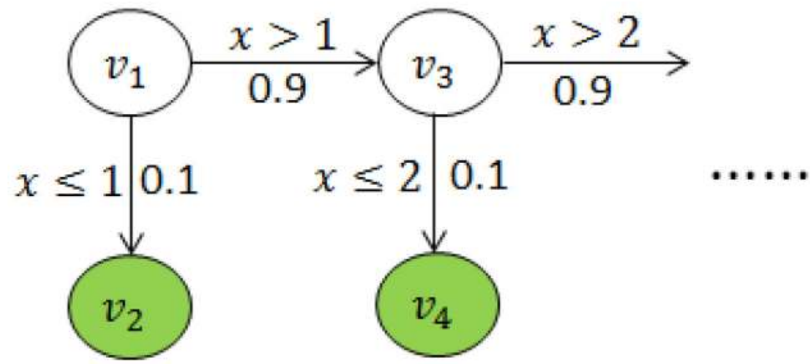


Figure 3.
SquID Example

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript

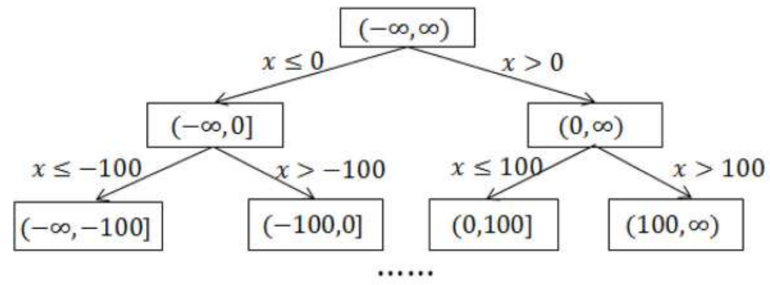
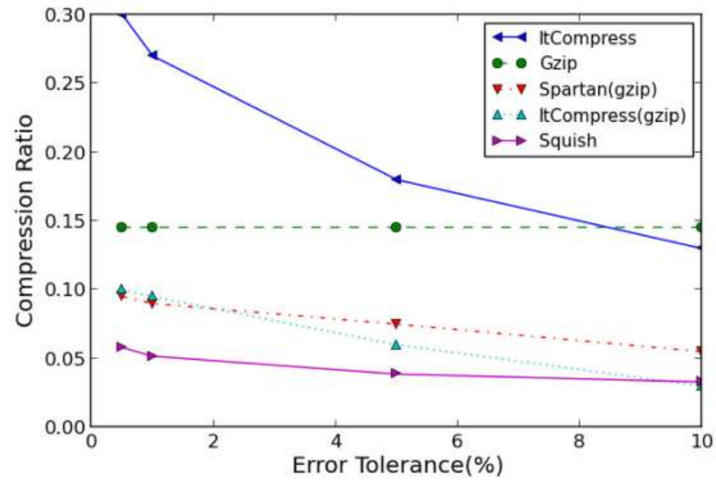
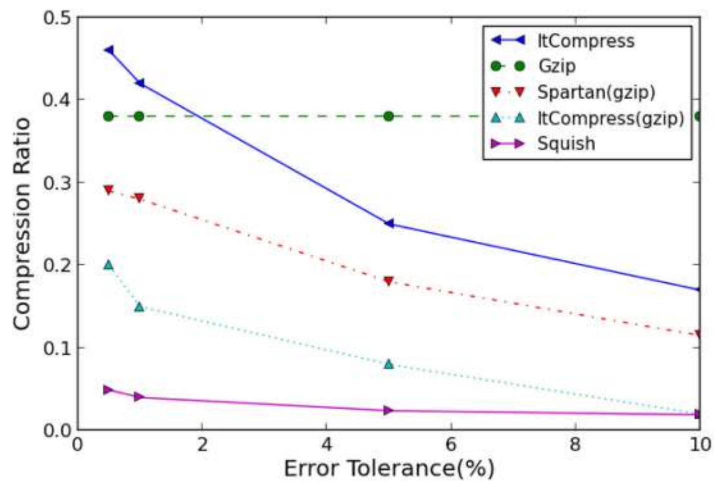


Figure 4.
SquID for numerical attributes

Author Manuscript Author Manuscript Author Manuscript Author Manuscript



(a) Forest Cover



(b) Corel

Figure 5.
Error Threshold vs Compression Ratio

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

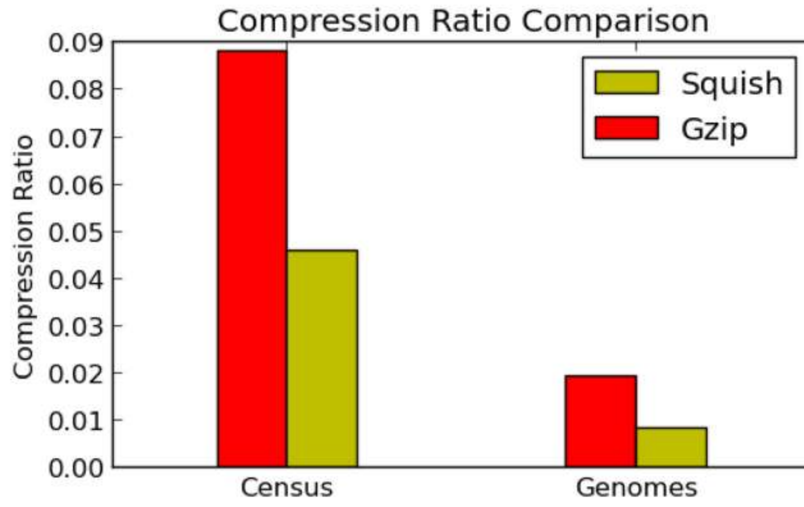


Figure 6.
Compression Ratio Comparison

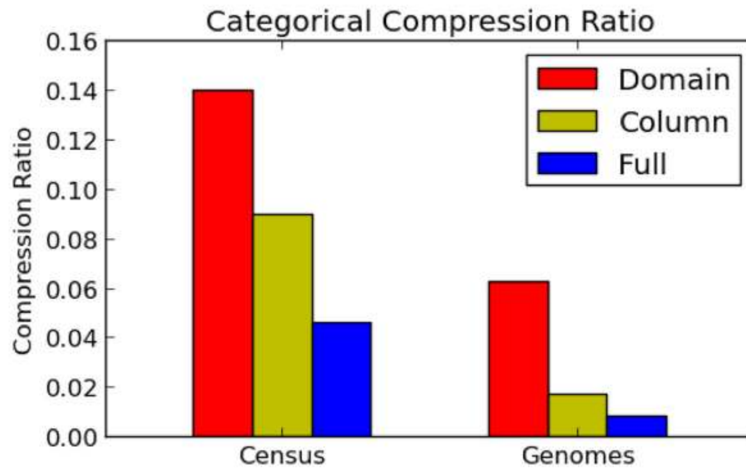


Figure 7.
Compression Ratio Comparison

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

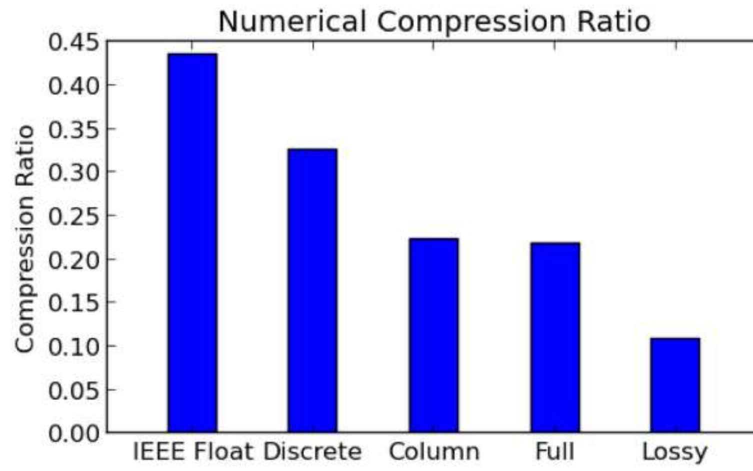


Figure 8.
Compression Ratio Comparison

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

Table 1

Features of Our System contrasted with Prior Work

System	AC	NA	LC	UDA
SQUISH	Y	Y	Y	Y
Spartan [3]	Y	Y	Y	N
ItCompress [8]	Y	Y	Y	N
Davies and Moore [6]	Y	N	N	N
Raman and Swart [12]	N	N	N	N

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

Table 2

Functions need to be implemented for SquID Template

Function	Description
IsEnd	Return whether the current node is a leaf node.
GenerateBranch	Return the number of branches and the probability distribution associated with them.
GetBranch	Given an attribute value, return which branch does the value belong to.
ChooseBranch	Set the current node to another node at next level according to the given branch index.
GetResult	If the current node is a leaf node, return the representative attribute value of this node.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Table 3

Running Time of Different Components

	Forest Cov.	Corel	Census	Genomes
Struct. Learning	5.5 sec	2.5 sec	20 min	N/A
Param. Tuning	140 sec	15 sec	100 min	40 min
Compression	48 sec	6 sec	6 min	50 min
Writing to File	7 sec	2 sec	40 sec	7 min
Decompression	53 sec	7.5 sec	6 min	50 min

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Table 4

Sensitivity of the Structure Learning

No. of Exp.	1	2	3	4	5
Comp. Ratio	0.0460	0.0472	0.0471	0.0468	0.0476

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Table 5

Sensitivity to Number of Tuples

Number of Tuples	1000	2000	5000
Comp. Ratio	0.0474	0.0460	0.0427