Squeezer: An efficient algorithm for clustering categorical data

Article *in* Journal of Computer Science and Technology · September 2002

DOI: 10.1007/BF02948829 · Source: DBLP

Zengyou He
Dalian University of Technology
124 PUBLICATIONS **4,119** CITATIONS
SEE PROFILE

Shengchun Deng
Harbin Institute of Technology
61 PUBLICATIONS **2,272** CITATIONS
SEE PROFILE

Squeezer: An Efficient Algorithm for Clustering Categorical Data

HE Zengyou (何增有), XU Xiaofei (徐晓飞) and DENG Shengchun(邓胜春)

Department of Computer Science and Engineering, Harbin Institute of Technology
Harbin 150001, P.R. China

E-mail: zengyouhe@yahoo.com; {xiaofei,dsc}@hope.hit.edu.cn

Received November 20, 2001; revised March 4, 2002.

Abstract This paper presents a new efficient algorithm for clustering categorical data, *Squeezer*, which can produce high quality clustering results and at the same time deserve good scalability. The *Squeezer* algorithm reads each tuple t in sequence, either assigning t to an existing cluster (initially none), or creating t as a new cluster, which is determined by the similarities between t and clusters. Due to its characteristics, the proposed algorithm is extremely suitable for clustering data streams, where given a sequence of points, the objective is to maintain consistently good clustering of the sequence so far, using a small amount of memory and time. Outliers can also be handled efficiently and directly in *Squeezer*. Experimental results on real-life and synthetic datasets verify the superiority of *Squeezer*.

Keywords clustering, categorical data, data stream, data mining

1 Introduction

Clustering is an important KDD technique with numerous applications, such as marketing and customer segmentation. Clustering typically groups data into sets in such a way that the intra-cluster similarity is maximized while the inter-cluster similarity is minimized. Many efficient clustering algorithms, such as ROCK[1], C^2P[2], DBSCAN[3], BIRTH[4], CURE[5], CHAMELEON[6], WaveCluster[7] and CLIQUE[8], have been proposed by the database research community.

Most previous clustering algorithms focus on numerical data whose inherent geometric properties can be exploited naturally to define distance functions between data points. However, many of the data in databases are categorical, where attribute values cannot be naturally ordered as numerical values. An example of categorical attribute is *shape* whose values include *circle*, *rectangle*, *ellipse*, etc. Due to the special properties of categorical attributes, the clustering of categorical data seems more complicated than that of numerical data.

In this paper, we present *Squeezer*, a new clustering algorithm for categorical data. The basic idea of *Squeezer* is simple. *Squeezer* repeatedly reads tuples from a dataset one by one. When the first tuple arrives, it forms a cluster alone. The consequent tuples are either put into existing clusters or rejected by all existing clusters to form a new cluster by given a similarity function defined between a tuple and a cluster.

The *Squeezer* algorithm only makes one scan over the dataset, thus, is highly efficient for disk resident datasets where the I/O cost becomes the bottleneck of efficiency.

The main objective of *Squeezer* is the combination of efficiency and scalability. Experimental results show that *Squeezer* achieves both high quality clustering results and scalability. In summary, the main contributions of this paper are:

• A novel algorithm for clustering categorical data, *Squeezer*, achieves both efficiency and high quality clustering results.

This work was supported by the National Natural Science Foundation of China (Grant No. 60084004) and the IBM AS/400 Research Fund.

• The algorithm is suitable for clustering data streams, where given a sequence of points, the objective is to maintain consistently good clustering of the sequence so far, using a small amount of memory and time.

• Outliers can be handled efficiently and directly.

• The algorithm does not require the number of desired clusters as an input parameter. This is very important for the user who usually does not know this number in advance. The only parameter to be pre-specified is the value of similarity between the tuple and the cluster, which incorporates the user's expectation that how close the tuples in a cluster should be.

The rest of this paper is organized as follows. In Section 2, definitions used in our new clustering model are formalized and illustrative examples are given. Section 3 presents the detailed algorithms and basic analysis. Experimental results are given in Section 4. Section 5 discusses the relationship between our work and some related work. Finally, Section 6 concludes the paper.

2 Definitions

In this section, we formalize the definitions needed in our new clustering model and introduce other concepts used in the remainder of the paper.

Let $A_1, \ldots, A_m$ be a set of categorical attributes with domains $D_1, \ldots, D_m$ respectively. Let the dataset D be a set of tuples where each tuple t: $t \in D_1 \times \cdots \times D_m$. Let *TID* be the set of unique IDs of all tuples. For each $tid \in TID$, the attribute value for A_i of the corresponding tuple is represented as $val(tid, A_i)$.

Definition 1 (Cluster). $Cluster = \{tid | tid \in TID\}$ is a subset of *TID*.

Informally, a cluster produced by clustering algorithms is a subset of all the tuples in the dataset. The *Cluster* presented in Definition 1 is semantically equivalent to its traditional meaning for the element in *TID* can identify each tuple uniquely.

Definition 2. *Given a Cluster C, the set of attribute values on A_i with respect to C is defined as*: $VAL_i(C) = \{val(tid, A_i) | tid \in C\}$.

In fact, the set of attribute values defined here is the result of projecting tuples in C on A_i. Note that the set $VAL_i(C)$ contains distinct attribute values.

Definition 3. *Given a Cluster C, and $a_i \in D_i$, then the support of a_i in C with respect to A_i is defined as*: $Sup(a_i) = |\{tid | tid.A_i = a_i\}|$.

According to Definition 3, the support of an attribute value is the number of tuples in the *Cluster* containing the value. It can be 0 when no tuple in the *Cluster* contains this value. The attribute value with large support indicates that the probability of its appearance is bigger than others.

Definition 4 (Summary). *Given a Cluster C, the Summary for C is defined as*:

$$Summary = \{\, VS_i | 1 \leq i \leq m\} \text{ where } VS_i = \{(a_i, Sup(a_i)) | a_i \in VAL_i(C)\}.$$

Intuitively, the *Summary* of a *Cluster* contains summary information about the *Cluster*. In general, each *Summary* consists of m elements, where m is the number of attributes. The element in a *Summary* is the set of pairs of attribute values and their corresponding supports. We will show later that information contained in a *Summary* is sufficient to compute the similarity between a tuple and *Cluster*.

Definition 5 (Cluster Structure, CS). *Given a Cluster C, the Cluster Structure (CS) for C is defined as*: $CS = \{Cluster, Summary\}$.

The *Cluster Structure* is the main data structure used in the *Squeezer* algorithm. We use the *Cluster* to store classified tuples and the *Summary* to compute similarities.

To illustrate the above definitions, consider the example given in Table 1.

Example 1. Let us consider the following dataset representing the assignment of employees to departments. The dataset consists of 7 tuples with 5 attributes.

For briefness, attributes EmpNum, DepNum, Year, DepName and Mgr are renamed with A, B, C, D, and E respectively, and all the attributes are considered to be categorical.

Table 1. A Table of Employees

Tuple ID	EmpNum	DepNum	Year	DepName	Mgr
1	1	1	85	Biochemistry	5
2	1	5	94	Admission	12
3	2	2	92	DB	2
4	3	2	98	DB	2
5	4	3	98	AI	2
6	5	1	98	Biochemistry	2
7	6	5	88	Admission	12

Given a *Cluster* $C = \{3, 4, 5\}$, we can get a $Summary = \{\{(2,1),(3,1),(4,1)\},\{(2,2),(3,1)\},\{(92,1),(98,2)\},\{(DB,2),(AI,1)\},\{(2,3)\}$. The Cluster Structure (CS) is $CS = \{Cluster, Summary\}$.

Definition 6 (Similarity Function). *Given a Cluster C and a tuple t with $tid \in TID$, the similarity between C and t is defined as*:

$$Sim(C, tid) = \sum_{i=1}^{m} \Big(\frac{Sup(a_i)}{\sum_j Sup(a_j)}\Big) \text{ where } tid.A_i = a_i \text{ and } a_j \in VAL_i(C).$$

From Definition 6, it is clear that the similarity is statistics based. In other words, if the similarity between a tuple and an existing cluster is big enough, then it is more probable that the tuple belongs to this cluster. In our algorithm, this measure is used to determine whether the tuple should be put into the cluster or not.

Example 2. Continuing with Example 1, the similarity between the *Cluster* $C = \{3, 4, 5\}$ and the tuple with $tid = 6$ can be computed using Definition 6 as:

$$Sim(\{3,4,5\},6) = \sum_{i=1}^{5} \Big(\frac{Sup(a_i)}{\sum_j Sup(a_j)}\Big) = \Big(\frac{0}{1+1+1} + \frac{0}{2+1} + \frac{2}{2+1} + \frac{0}{2+1} + \frac{3}{3}\Big) \approx 1.67$$

3 Algorithms and Basic Analysis

In this section, we will describe our core algorithms: the *Squeezer* algorithm and the *d-Squeezer* algorithm. The *d-Squeezer* is a variant of *Squeezer* to handle dataset with large volume.

3.1 Overview

The *Squeezer* algorithm has n tuples as input and produces clusters as final results. Initially, the first tuple in the database is read in and a *Cluster Structure* (*CS*) is constructed with $C = \{1\}$. Then, the subsequent tuples are read iteratively.

For each tuple, by our similarity function, we compute its similarities with all existing clusters, which are represented and embodied in the corresponding *CSs*. The largest value of similarity is selected out. If it is larger than the given threshold, denoted as s, the tuple is put into the cluster that has the largest value of similarity. The *CS* is also updated with the new tuple. If the above condition does not hold, a new cluster must be created with this tuple.

The algorithm continues until all tuples in the dataset are traversed.

3.2 Details

The *Squeezer* algorithm is presented in Fig.1. It accepts as input the dataset D and the value of the desired similarity threshold. The algorithm fetches tuples from D iteratively.

Initially, the first tuple is read in, and the sub-function *addNewClusterStructure*() is used to establish a new *Clustering Structure*, which includes *Summary* and *Cluster* (Steps 3 – 4).

For each subsequent tuple, the similarity between an existing *Cluster* C and the tuple is computed using sub-function *simComputation*() (Steps 6 – 7). We get the maximal value of similarity (denoted by *sim_max*) and the corresponding index of *Cluster* (denoted by *index*) from the above computing results (Steps 8 – 9). Then, if the *sim_max* is larger than the input threshold s, sub-function *addTupleToCluster*() will be called to assign the tuple to the selected *Cluster* (Steps 10 – 11). If it is not the case, the sub-function *addNewClusterStructure*() will be called to construct a new *CS* (Steps 12 – 13). Finally, outliers will be handled (Step 15) and the clustering results will be labeled on the disk (Step 16).

In the following, we will give descriptions on the sub-functions used in the *Squeezer* algorithm.

The sub-function *addNewClusterStructure*() is presented in Fig.2. It uses the new tuple to initialize *Cluster* and *Summary*, and then a new *CS* is created. The sub-function *addTupleToCluster*() is shown in Fig.3, which updates the specified *CS* with new tuple. Fig.4 gives a glance at the sub-function *simComputation*(), which makes use of information stored in the *CS* to get the statistics based similarity.

```
Algorithm Squeezer $(D, s)$
Begin
1.  while ($D$ has unread tuple) {
2.      tuple = getCurrentTuple ($D$)
3.      if (tuple.tid == 1) {
4.          $addNewClusterStructure(tuple.tid)$}
5.      else {
6.          for each existing cluster $C$
7.              $simComputation(C$, tuple)
8.          get the max value of similarity: $sim\_max$
9.          get the corresponding Cluster Index: $index$
10.         if $sim\_max \geq s$
11.             $addTupleToCluster(tuple, index)$
12.         else
13.             $addNewClusterStructure(tuple.tid)$}
14. }
15. $handleOutliers()$
16. $outputClusteringResult()$
End
```

Fig.1. Squeezer algorithm.

```
Sub_Function $addNewClusterStructure(tid)$
1. $Cluster = \{tid\}$
2. for each attribute value $a_i$ on $A_i$
3.     $VS_i = (a_i, 1)$
4.     add $VS_i$ to $Summary$
5. $CS = \{Cluster, Summary\}$
```

Fig.2. Sub-function *addNewClusterStructure*().

```
Sub_Function $addTupleToCluster(tuple, index)$
1. $Cluster = Cluster \cup \{tuple.tid\}$
2. for each attribute value $a_i$ on $A_i$
3.     $VS_i = (a_i, Sup(a_i) + 1)$
4.     add $VS_i$ to $Summary$
5. $CS = \{Cluster, Summary\}$
```

Fig.3. Sub-function *addTupleToCluster*().

```
Sub_Function $simComputayion(C, tuple)$
1. defin $sim = 0$
2. for each attribute value $a_i$ on $A_i$
3.     $sim = sim + probability\ of\ a_i\ on\ C$
4. return sim
```

Fig.4. Sub-function *simComputation*().

As to the sub-function *outputClusteringResult*(), it simply outputs the *Clusters* stored in *CS*s and writes them on to disk.

As pointed out in Section 1, the *Squeezer* algorithm can handle outliers efficiently and directly. In the end of the algorithm, we will get some *Clusters*. In the sub-function *handleOutlier*(), we can just ignore extremely small *clusters* as outliers and discard them. The intuitive reason behind is that in our clustering model, tuples in the smaller clusters are more likely to be outliers for they are more dissimilar with respect to a large portion of tuples in the dataset. Related research on outlier detection has empirically verified the effectiveness of this method[9].

3.3 Properties

In this section, we will exploit some interesting properties of *Squeezer*. These properties guarantee that *Squeezer* can produce high quality clustering results.

Definition 7. *For two tuples T_1, T_2 in dataset D, the similarity between them is defined as follows*: $sim(T_1, T_2) = |\{A_i | T_1 \cdot A_i = T_2 \cdot A_i,\ 1 \leq i \leq m\}|$.

Definition 8. *For a set of tuples T and a single tuple T_1, for $\forall T_i \in T$, the average similarity between T_1 and T_i is defined as follows*:

$$avg\text{-}sim(T_1, T_i) = \sum_i sim(T_1,\ T_i)/|T|.$$

The definition of *average similarity* captures the spirit that the similarity between tuples can be considered from a global point of view. In this way, the new concept of similarity incorporates global information about the other tuples, not only the specified two tuples. The extensions to the concept of similarity between tuples guarantee that *Squeezer* can produce more meaningful clusters.

Lemma 1. *Let s be an integer. Let C be a Cluster produced by Squeezer(D, s). Suppose $|C| = N$ and the tuples in C are arranged in the order of their insertion. Then, it is the case that*:

$$\textit{avg-sim}\ (T_N, T_i) \geq s \text{ where } i \text{ from } 1 \text{ to } N-1 \text{ and set } T = C - \{T_N\}.$$

Proof (Sketch). From the point of view of *Squeezer*, the tuple T_N is added to C because the following inequality holds:

$$\textit{Sim}(C - \{T_N\}, T_N) = \sum_{i=1}^{m} \Big(\frac{\textit{Sup}(a_i)}{\sum_j \textit{Sup}(a_j)} \Big) \geq s \tag{1}$$

while

$$\begin{aligned}\textit{avg-sim}(T_N, T_i) &= \sum_i \ \textit{sim}(T_N, T_i)/|C - \{T_N\}| \\ &= \sum_i \sum_{j=1}^{m} |a_{ji}|/|C - \{T_N\}| \quad //a_{ji} \in \{a | a = T_1.A_i = T_2.A_i,\ 1 \leq i \leq m\} \\ &= \sum_{j=1}^{m} \sum_i \frac{|a_{ji}|}{|C - \{T_N\}|} = \sum_{j=1}^{m} \frac{\sum_i |a_{ji}|}{|C - \{T_N\}|} = \sum_{j=1}^{m} \Big(\frac{\textit{Sup}(a_j)}{\sum_i \ \textit{Sup}(a_i)} \Big) \end{aligned} \tag{2}$$

From (1) and (2), $\textit{avg-sim}(T_N, T_i) \geq s$ holds. □

The interpretation of Lemma 1 is as follows. During the clustering process of *Squeezer*, only those tuples whose *average similarity* with other tuples in C exceeds the threshold s can be added to cluster C. It indicates that the tuple is put into the "best" cluster by considering *average similarity*.

Lemma 2. *Let s be an integer. Let C be a Cluster produced by Squeezer(D, s). Suppose $|C| = N$ and tuples in C are arranged in the order of their insertion. Then, it is the case that*:

$$\textit{avg-sim}(T_j, T_i) \geq s \text{ where } i \text{ from } 1 \text{ to } j-1 \text{ and } j = i+1, \text{ set } T = \{T_i | 1 \leq i \leq j-1\}.$$

Theorem 1 (A General Lower Bound on Average Similarity). *Let s be an integer. Let C be a Cluster produced by Squeezer (D, s). Suppose $|C| = N$ and tuples in C are arranged in the order of their insertion. Then, it is the case that*:

$$\textit{avg-sim}\ (T_i, T_j) \geq s(i-1)/(N-1) \text{ where } i \neq j \text{ and } 1 \leq i, j \leq N, \ \text{ set } T = C - \{T_i\}.$$

Proof.

$$\begin{aligned}\textit{avg-sim}(T_i, T_j) &= \sum_{j=1}^{N} \textit{sim}(T_i, T_j)/(N-1) \\ &= \Big(\sum_{j=1}^{i-1} \ \textit{sim}(T_i, T_j) + \sum_{j=i+1}^{N} \ \textit{sim}(T_i, T_j) \Big) \Big/ (N-1) \\ &\geq (i-1)s/(N-1) + \sum_{j=i+1}^{N} \ \textit{sim}(T_i, T_j)/(N-1) \geq (i-1)s/(N-1)\end{aligned}$$

Theorem 1 gives a general lower bound on the *average similarity* for every two tuples in a cluster. It is clear that the lower bound of *average similarity* of tuples is dependent on the order of its insertion to the cluster. Those tuples added to the cluster later tend to have a larger lower bound than others.

Theorem 2. *Let s_1, s_2 be two integers and $s_1 > s_2$. Let k_1 be the number of clusters produced by Squeezer(D, s_1) and k_2 be the number of clusters produced by Squeezer(D, s_2). Then, $k_1 \geq k_2$ holds.*

Proof (Sketch). First, it should be noted that with these two different thresholds, the tuples in the dataset D are read in the same order. The only thing to be proved is that the number of times of sub-function *addNewClusterStructure*() called when threshold equals s_1 is at least *not* less than that of s_2.

Suppose there are N tuples in the dataset D, in the clustering process of *Squeezer*, Step 10 will be executed N times to compare the maximal value of similarity with the specified threshold. Then, we consider the execution of *Squeezer* as a sequence of length N from the point of view of the threshold. For instance, when the threshold is set to s_1, the sequence will be: $s_1, s_1, \ldots, s_1$. If in different steps we set different values to the threshold, we can give a series of sequences as the following:

$$\begin{cases} s_1, s_1, s_1, \ldots, s_1, s_1, \boldsymbol{s_1} & (1) \\ s_1, s_1, s_1, \ldots, s_1, \boldsymbol{s_1}, \boldsymbol{s_2} & (2) \\ s_1, s_1, s_1, \ldots, s_1, \boldsymbol{s_2}, s_2 & (3) \\ \cdots & (\cdots) \\ s_1, \boldsymbol{s_1}, s_2, \ldots, s_2, s_2, s_2 & (\text{N-2}) \\ \boldsymbol{s_1}, \boldsymbol{s_2}, s_2, \ldots, s_2, s_2, s_2 & (\text{N-1}) \\ \boldsymbol{s_2}, s_2, s_2, \ldots, s_2, s_2, s_2 & (\text{N}) \end{cases}$$

The first sequence indicates that in the whole lifetime of *Squeezer*, the threshold is set to s_1, while the second sequence says that for the last tuple the threshold is set to s_2. The explanations of other sequences are similar. It is easy to find that any adjacent sequences have the same elements in all positions except one. We use $C_1, C_2, \ldots, C_N$ to represent the clusters produced by *Squeezer* with corresponding sequences of thresholds. Obviously, $k_1 = |C_1|$ and $k_2 = |C_N|$.

For $s_1 > s_2$, from the Steps 10 – 13 in the algorithm *Squeezer* and the properties of these sequences, the following statements hold:

$$\begin{cases} |C_1| \geq |C_2| & (1) \\ |C_2| \geq |C_3| & (2) \\ \cdots & (\cdots) \\ |C_{N-1}| \geq |C_N| & (\text{N-1}) \end{cases}$$

Then, $k_1 = C_1 \geq k_2 = C_N$ holds.

The interpretation of Theorem 2 is as follows. Intuitively, with the increase of threshold s, the number of clusters produced by *Squeezer* will increase; at least it will not decrease. In practice, when the threshold s reaches its maximal value (the number of attributes), the result produced by *Squeezer* will be that every single tuple forms a cluster, when no duplicates exist.

3.4 Handling Large Databases: *d*-Squeezer

During the clustering process of *Squeezer*, our main task is to maintain and update several *CS*s. However, when the size of dataset is extremely large, the *Clusters* in *CS* will occupy a large amount of main memory. To handle large volume data, we propose *d-Squeezer*, an alternative of *Squeezer*.

The only difference between *d-Squeezer* and *Squeezer* is that: in *d-Squeezer*, instead of retaining the *Cluster* in *CS* in main memory, we write the *Cluster* identifier of each tuple back to the file immediately when it is assigned to a *Cluster*. Only the *Summary* in a *CS* and a *Counter* for the size of *Cluster* will be held in the main memory. For the *Summary* and *Counter* can be relatively small in size, they can be retained in main memory even in case that the database is very large. This property enables *d-Squeezer* to handle very large databases.

As stated in Definition 6, the similarity between a tuple and the cluster is computed as:

$$\sum_{i=1}^{m}\left(\frac{Sup(a_i)}{\sum_j Sup(a_j)}\right).$$

It is easy to get that $\sum_j Sup(a_j) = |C|$. To reduce the computation time, we will not compute $\sum_j Sup(a_j)$ and replace it with the size of *Cluster* fetched from *Counter* directly.

Fig.5 and Fig.6 describe the changed sub-functions of *addNewClusterStructure*() and *addTupleTo-Cluster*().

```
Sub_Function addNewClusterStructure(tid)
1. Write tid to Cluster on disk
2. Counter++
3. for each attribute value $a_i$ on $A_i$
4.     $VS_i = (a_i, 1)$
5.     add $VS_i$ to Summary
6. CS = { Summary, Counter}
```

Fig.5. *addNewClusterStructure*() for *d-Squeezer*.

```
Sub_Function addTupletoCluster(tuple, index)
1. Write tuple.tid to Cluster on disk
2. Counter++
1. for each attribute value $a_i$ on $A_i$
2.     $VS_i = (a_i,\ Sup\ (a_i) + 1)$
3.     add $VS_i$ to Summary
4. CS = { Summary, Counter}
```

Fig.6. *addTupleToCluster*() for *d-Squeezer*.

Instead of adding the *tid* of a tuple to the corresponding *Cluster* maintained in the main memory, we write the information onto the disk. As shown in Fig.5 and Fig.6, the *CS* will only contain *Summary* and the size of *Cluster* (denoted as *Counter*).

It is obvious that the results of clustering produced by *Squeezer* and *d-Squeezer* are the same in spite of the differences between them.

3.5 Time and Space Complexities

Worst-case analysis: The time and space complexities of the *Squeezer* algorithm depend on the size of dataset (n), the number of attributes (m), the number of the *CSs* and the size of every *CS*. To simplify the analysis, we assume that the final number of clusters is k, and every attribute has the same number of distinct attribute values, p. Then, in the worst case, from Subsection 3.2, we can get that the algorithm has time complexity $O(n * k * p * m)$ and space complexity $O(n + k * p * m)$.

Practical analysis: As pointed out in [10], categorical attributes usually have *small* domains. Typical categorical attribute domains considered for clustering consist of less than a hundred or, rarely, a thousand attribute values. An important implication of the compactness of categorical domains is that the parameter, p, can be regarded to be very small. The use of hashing technique in the search of attribute values in *Summary* can also reduce the impact of p. So, in practice, the time complexity of *Squeezer* can be expected to be $O(n * k * m)$.

The above analysis shows that the time complexity of *Squeezer* is linear with the size of dataset, the number of attributes and the final number of clusters, which make this algorithm deserve good scalability.

4 Experimental Results

This section contains the experimental results about the performance of *Squeezer*. Both the quality of the clustering results and the efficiency are examined. We ran *Squeezer* on real-life as well as synthetic datasets. The use of real-life datasets is to compare the quality of the clustering results produced by *Squeezer* with other algorithms, such as ROCK. The synthetic datasets are used to primarily demonstrate the scalability of *Squeezer* and *d-Squeezer*.

Our algorithms were implemented in Java. All experiments were conducted on a Pentium Ⅲ-600 machine with 128M of RAM and running Windows 2000 Server.

4.1 Quality of Clustering Results with Real-Life Datasets

In this experiment, we compare *Squeezer* with ROCK, for ROCK can produce good clustering results. We experimented with two real-life datasets: the Congressional Voting dataset and the Mushroom dataset, which were obtained from the UCI Machine Learning Repository[11] and used in ROCK[1]. Now we will give a brief introduction to these two datasets.

• **Congressional Votes:** It is the United States Congressional Voting Records in 1984. Each tuple represents one Congressman's votes on 16 issues. All attributes are Boolean with Yes (denoted as y) or No (denoted as n) values. A classification label of Republican or Democrat is provided with each tuple. The dataset contains 435 tuples with 168 Republicans and 267 Democrats.

• **Mushroom:** The mushroom dataset has 22 attributes with 8,124 tuples. Each tuple records physical characteristics of a single mushroom. A classification label of poisonous or edible is provided for each tuple. The numbers of edible and poisonous mushrooms in the dataset are 4,208 and 3,916, respectively.

Table 2 contains the initial results of running on congressional voting data using *Squeezer* with s set to 10. The clusters in Table 2 are the original results produced by *Squeezer* without outlier post-processing. If we regard clusters with a size less than 10 as outliers and remove them, only the set {1, 2, 3, 5, 11, 13, 18} of clusters will be retained. After removing outliers, the results of clusters produced by *Squeezer* and ROCK (we get the results of ROCK given by [1]) are described in Table 3.

Table 2. Initial Results on Congressional Voting Data by *Squeezer*

Cluster No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Republicans	**124**	**0**	**3**	0	**15**	0	0	0	1	2	**1**	0	**7**	0
Democrats	**9**	**14**	**134**	3	**1**	8	5	2	1	1	**32**	3	**9**	9
Cluster No.	15	16	17	*18*	19	20	21	22	23	24	25	26	27	28
Republicans	7	0	2	**0**	4	0	0	0	0	2	0	0	0	0
Democrats	0	4	0	**10**	0	1	5	4	4	1	3	1	1	2

As Table 3 illustrates, both *Squeezer* and ROCK can identify clusters with the majority of Republicans or Democrats. Due to the elimination of outliers, the sum of the sizes of clusters produced by ROCK and *Squeezer* is not equal to the size of the original input size. From the summary information of Table 3 we can conclude that both algorithms can produce clusters of high quality.

We also compare our algorithm with ROCK using the mushroom dataset. Table 4 gives the results of ROCK and *Squeezer* with s set to 16.

Table 3. Results on Congressional Voting Data by *Squeezer* and ROCK

ROCK (Results Reported in [1])		
Cluster No.	*No. of Republicans*	*No. of Democrats*
1	144	22
2	5	201
Squeezer		
Cluster No.	*No. of Republicans*	*No. of Democrats*
1	124	9
2	0	14
3	3	134
4	15	1
5	1	32
6	7	9
7	0	0

As shown in Table 4, both algorithms can find pure clusters except one (cluster 15 in ROCK, cluster 14 in *Squeezer*) in the sense that mushrooms in each cluster were either all edible or all poisonous. However, cluster 14 in *Squeezer* contains *only one* edible mushroom with all the others being poisonous. Cluster 15 in ROCK contains 32 edible mushrooms and 72 poisonous mushrooms. In this case, *Squeezer* performed better than ROCK and it was shown that *Squeezer* can produce high quality clusters.

Table 4. Results with Mushroom Data by *Squeezer* and ROCK

ROCK (Results Reported in [1])					
Cluster No.	*No. of Edible*	*No. of Poisonous*	*Cluster No.*	*No. of Edible*	*No. of Poisonous*
1	96	0	12	48	0
2	0	256	13	0	288
3	704	0	14	192	0
4	96	0	15	**32**	**72**
5	768	0	16	0	1,728
6	0	192	17	288	0
7	1,728	0	18	0	8
8	0	32	19	192	0
9	0	1,296	20	16	0
10	0	8	21	0	36
11	48	0			
Squeezer					
Cluster No.	*No. of Edible*	*No. of Poisonous*	*Cluster No.*	*No. of Edible*	*No. of Poisonous*
1	0	256	13	48	0
2	512	0	14	**1**	**72**
3	768	0	15	48	0
4	96	0	16	0	32
5	96	0	17	0	8
6	192	0	18	0	859
7	1,728	0	19	192	0
8	0	1,296	20	288	0
9	0	192	21	0	36
10	0	288	22	31	0
11	192	0	23	0	8
12	0	869	24	16	0

4.2 Tuning the Parameter of *Squeezer*

The similarity threshold s is the only parameter needed in the *Squeezer* algorithm, which can affect the results of clustering and the speed of algorithm. As pointed out in Subsection 3.3, different values of s may produce different numbers of clusters. In this section, we will give some empirical results to shown how it can affect the *Squeezer* algorithm with the Congressional Voting dataset and the Mushroom dataset

Since the classification label is provided for each tuple in the two datasets, to test the impact of changing parameter s on the quality of clustering, we will use the percentage of misclassified tuples as an assessment. Fig.7 describes the number of produced clusters with changing s, and Fig.8 shows how the percentage of misclassified tuples changes when s increases.

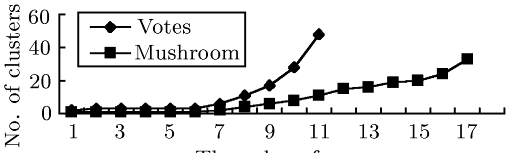

Fig.7. Number of clusters with different s values.

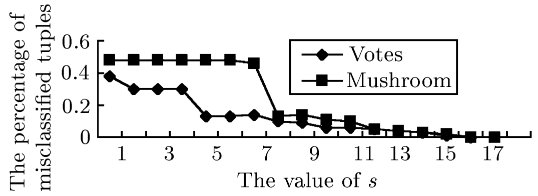

Fig.8. The percentage of misclassified tuples with different s values.

Just as we have pointed out in Theorem 2, with the increase of s, the number of clusters produced by *Squeezer* increases, and the experimental results given in Fig.7 confirm our idea. At the same time, as shown in Fig.8, the increase of s will result in the improvement of accuracy of clustering. However, if the number of clusters is too large, it prevents us from getting meaningful clusters. On the other hand, if the value of s is small, all tuples are put into the same cluster, which will result in little usefulness of the *Squeezer* algorithm.

Furthermore, with the increase of the number of clusters, the execution time of *Squeezer* will increase significantly, as presented in Fig.9.

Fig.9 gives the results of execution time when s increases. As we pointed out in Subsection 3.5, the *Squeezer* algorithm is linear with the number of clusters. Thus, the increase of s will result in the increase of execution time.

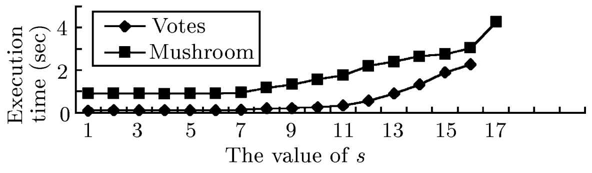

Fig.9. The execution time with different s values.

The above experiments show that the choice of s can affect both the quality of clustering and execution time of *Squeezer*. Thus, choosing a proper value for the parameter s is one of important tasks that must be considered in the *Squeezer* algorithm.

To solve this problem, in our current implementation, the following strategy is adopted.

From the analysis of Subsection 3.3, for a given similarity threshold s, it is expected that the *average similarity* between any two tuples in the same cluster is larger than s. Therefore, this parameter can be specified as the user's expectation that how close the tuples in a cluster should be in the case that the user is familiar with the dataset. However, it is often the case that the properties of the target dataset to be analyzed are not known in advance. We use the sampling technique to get a candidate value of threshold, which is described as the following steps:

1) Get a sample of the whole dataset, denoted as S.

2) For every pair of tuples in S, compute the similarity between them by Definition 7.

3) Compute the average value of the similarities from Step 2, denoted as *avg_value*.

4) Set $s = avg\_value + 1$ or $s = avg\_value + 2$.

To verify the effectiveness of our method for the automatic assignment of an appropriate value for the input parameter, we ran experiments on the Congressional Voting dataset and the Mushroom dataset.

For each dataset, we randomly chose 1/10 of the dataset as a sample and for each sample we ran 10 times to get an average value of *avg_value*. For the Congressional Voting dataset, we get $avg\_value = 8$. According to our heuristic, the similarity threshold can be set to 9 or 10. From our previous experimental results, it is clear that these two values can be regarded as good choices from the point of view of the number of clusters and clustering accuracy. For the Mushroom dataset, we get $avg\_value = 13$, thus, s will be 14 or 15. Similar to the analysis of Congressional Voting dataset, these automatically produced values are also "good" values.

4.3 Order Sensitivity

For the *Squeezer* algorithm to read tuples in sequence, in this section, we will test the sensitivity of our algorithm to the input sequence of tuples.

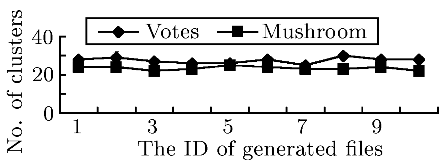

Fig.10. No. of clusters with different input sequences.

To find out how the input sequence of tuples affects the *Squeezer* algorithm, we ran experiments on the real life datasets of Mushroom and Congressional Voting. For each dataset, we produced 10 new datasets with each tuple placed randomly. Thus, by executing *Squeezer* on these datasets we can get results with different input sequences of tuples.

In the experiments, s was set to 10 for the datasets generated from Congressional Voting and 16 for Mushroom.

First, experiments were conducted to see the impact on the number of final clusters. As Fig.10

shows, the numbers of clusters produced by *Squeezer* with different input sequences are almost the same, which gives the evidence that the processing order of tuples does not have any major impact on the number of clusters.

With a careful analysis of the clustering results using different processing orders, it reveals that, for the same number of clusters, the clustering results are almost the same except for the movement of several tuples from one cluster to another. If the size of clusters changes, approximately, combining two or more clusters with larger size will give results for smaller size of clusters. Due to space limitation, we omit the details.

From the above experimental results, we can confidently assert that the *Squeezer* algorithm is robust with respect to input sequences of tuples.

4.4 Scalability with Synthetic Dataset

For the running time, experiments were carried out with synthetic datasets. Since the complexity of the ROCK algorithm is quadratic with the number of tuples in the database, it uses sampling to handle large datasets. The scalability of the algorithm is determined by the sample size, which makes the comparison on scalability between ROCK and *Squeezer* difficult. However, we believe that, in ROCK, to preserve the quality of clustering, the sample size must be approximately set to be large enough according to the database size. With the increase of sample size, scalability of ROCK will degrade since it is quadratic with the size of sample.

Thus, we compare *Squeezer* algorithm with CACTUS[10] algorithm and k-modes[12] algorithm, both of which have good scalabilities. The CACTUS algorithm scans the whole dataset twice. In the first scan summary information is collected and in the second scan clusters are produced. To make the comparison more convincing, we implemented the CACTUS in a more efficient way. In our implementation, in the first scan we only collected inter-attributes information and in the second scan nothing but clusters were labeled on the disk. Obviously, this implementation of CACTUS runs faster than the original one. For the k-modes algorithm①, we set the final number of clusters to be 2 to make it run faster and save the final results onto disk. For more details about CACTUS and k-modes refer to [10, 12].

To find out how the number of tuples affects the 4 algorithms, we ran a series of experiments with increasing numbers of tuples. The datasets were generated using a data generator, in which all possible values were produced with (approximately) equal probability. We set the number of tuples to 1 million, the number of attributes to 10 and the number of attribute values for each attribute to 10. Due to sparseness of generated datasets, we set s to 1.

Fig.11 shows the scalability of *Squeezer* and *d-Squeezer*, CACTUS, and k-modes while increasing the number of tuples from 1 to 10 million. When the number of tuples goes up to 3 million, the *Squeezer* runs out of memory.

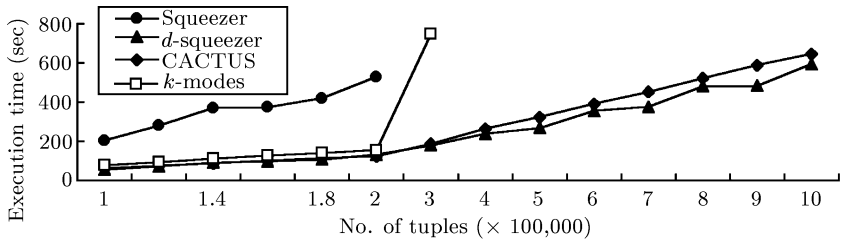

Fig.11. The execution time with different numbers of tuples.

To find out how the number of attributes affects these algorithms, we ran a series of experiments with increasing numbers of attributes. In Figs.12 and 13, the number of attributes is increased from

①The k-modes program used here is implemented in C and tested in Solaris 2.6 with 128M RAM, so we believe that it is faster than its implementation in JAVA.

10 to 100 and the number of tuples is fixed to 100,000.

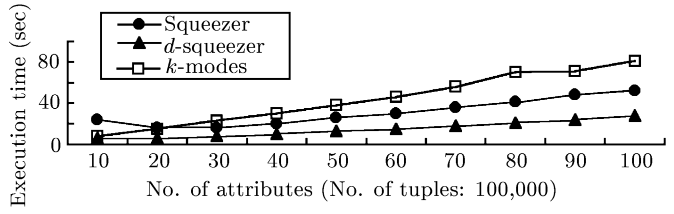

Fig.12. The execution time with different numbers of attributes (1).

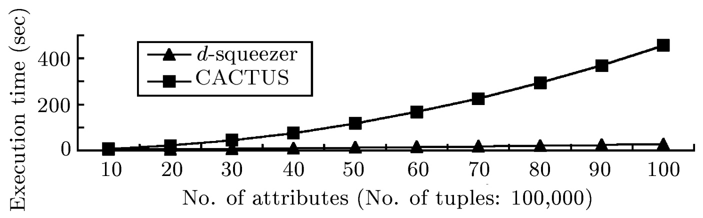

Fig.13. The execution time with different numbers of attributes (2).

The above experimental results demonstrate the scalability of *Squeezer* and *d-Squeezer* with respect to both the size of dataset and the number of dimensions. At the same time, the *d-Squeezer* algorithm outperforms the other algorithms in both cases.

5 Related Work

A few algorithms have been proposed in recent years for clustering categorical data[1,10,12–16]. In [15], the problem of clustering customer transactions in a market database is addressed. First, frequent itemsets are used to construct a weighted hyper-graph, where each frequent itemset is a hyper-edge and the weight of each hyper-edge is computed as the average of the confidence for all possible association rules that can be generated from the itemset. Then, a hyper-graph partitioning algorithm is used to partition the items such that the sum of weights of hyper-edge cut is minimized. Finally, customer transactions are assigned to the best item cluster according to items contained in each transaction and the item clusters. As pointed out in [1], this method is questionable for it makes the assumption that itemsets that define clusters are disjoint and have no overlap among them. This may not be true in practice since transactions in different clusters may have common items.

STIRR, an iterative algorithm based on non-linear dynamical systems, is presented in [13]. The approach used in [13] can be mapped to a certain type of non-linear systems. If the dynamical system converges, the categorical databases can be clustered. Another recent research[14] shows that the known dynamical systems cannot guarantee convergence, and proposes a revised dynamical system in which convergence can be guaranteed.

K-modes, an algorithm extending the k-means paradigm to the categorical domain, is introduced in [12]. New dissimilarity measures to deal with categorical data are used to replace means with modes, and a frequency based method is used to update modes in the clustering process to minimize the clustering cost function. This algorithm is scalable in terms of both the number of clusters and the number of tuples. However, as pointed out in [1], traditional partitioning algorithms are not suitable to categorical data, and the quality of clustering cannot be guaranteed.

In [10], the authors introduced a novel formalization of a cluster for categorical data by generalizing a definition of cluster for numerical data. A fast summarization based algorithm, CACTUS, is presented. CACTUS consists of three phases: *summarization*, *clustering* and *validation*. In the summarization phase, summary information about the dataset is collected. In the clustering phase, summary information is used to discover a set of candidate clusters. In the validation phase, the actual set of clusters is derived from the candidate clusters. In this method, the distinguishing subset assumption is made. However, this assumption does not always hold in practice.

ROCK, an adaptation of an agglomerative hierarchical clustering algorithm, is introduced in [1]. This algorithm starts with assigning each tuple to a separated cluster, and then clusters are merged repeatedly according to the closeness between clusters. The closeness between clusters is defined as the sum of the number of "links" between all pairs of tuples, where the number of "links" is computed as the number of common neighbors between two tuples. This link based method can produce good clustering results. Since the complexity of this algorithm is quadratic with the number of tuples in the database, sampling is used in it to handle large datasets. So, the scalability of the algorithm is determined by the sample size. To preserve the quality of clustering, the sample size must be approximately set to be large enough according to the database size. With the increase of database size, scalability of the algorithm will degrade.

In [16], the authors proposed the notion of *large item*. An item is *large* in a cluster of transactions if it is contained in a user specified fraction of transactions in that cluster. An allocation and refinement strategy, which has been adopted in partitioning algorithms such as k-means, is used to cluster transactions by minimizing the criteria function defined with the notion of large item.

The *similarity measure* used is a key aspect for the clustering algorithm, which can determine the quality of final clustering results. Therefore, we compare the *average similarity* in this paper with existing ones.

The definition of *average similarity* captures the spirit that the similarity between tuples can be considered from a global point of view. In this way, the new concept of similarity incorporates global information about the other tuples, rather than only the specified two tuples. The similarity measures used in [1, 10, 15, 16] also have the same advantage. In detail, [10, 15, 16] explicitly require that tuples in the same cluster should have *patterns* in common, and these *common patterns* can be described explicitly, for example, the *large item* in [16]. In contrast, the term *links* in [1] is similar with our *average similarity* in essence. Both of these two measures also require that tuples in the same cluster should have *patterns* in common. However, these *common patterns* cannot be described explicitly. For example, the *common neighbors* in [1] and the *average similarity* are not fixed in the cluster for different data points. From the experimental result for the Mushroom Dataset in Subsection 4.1, the likeness between clusters produced by ROCK and *Squeezer* verifies our analysis. Furthermore, despite these common features, the *average similarity* can be computed more efficiently than other measures; it guarantees that the *Squeezer* algorithm is better than other algorithms with respect to scalability.

6 Conclusions

In this paper, we consider the problem of clustering categorical data in large databases. An efficient algorithm named *Squeezer* is proposed, which can produce high quality clustering results and at the same time deserves good scalability.

Furthermore, the *Squeezer* algorithm can handle outliers efficiently and directly, which makes it robust to the effect of noise. Especially, this algorithm is suitable for clustering data streams. In the data stream model, data points can only be accessed in the order of their arrivals and random access is disallowed. And the space available to store information is supposed to be small compared with the huge size of unbounded streaming data points. Thus, the data mining algorithms on data streams are restricted to fulfill their tasks with only one pass over data sets and limited resources. The performance of an algorithm operating on a data stream is measured by the number of passes the algorithm must make over the stream[17]. As we have shown, the *Squeezer* algorithm needs to scan the database only once, and maintains some *Cluster Structure*s in the main memory with small space requirement. This perfect property qualifies it for the task of clustering data streams.

In the future work, we will revise *Squeezer* to make it more suitable for clustering data streams in the restricted data stream model. Automatic assignment of an appropriate value for the input parameter of *Squeezer* will be also further addressed.

Acknowledgements The authors wish to thank Dr. Longtao He for his help in debugging the code. We would also like to express our gratitude for Dr. Zhexue Huang of the University of Hong Kong for sending us helpful materials and the code of k-modes.

References

[1] Sudipto Guha, Rajeev Rastogi, Kyuseok Shim. ROCK: A robust clustering algorithm for categorical attributes. In *Proc. 1999 Int. Conf. Data Engineering*, Sydney, Australia, Mar., 1999, pp.512–521.

[2] Alexandros Nanopoulos, Yannis Theodoridis, Yannis Manolopoulos. C2P: Clustering based on closest pairs. In *Proc. 27th Int. Conf. Very Large Database*, Rome, Italy, September, 2001, pp.331–340.

[3] Ester M, Kriegel H P, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases. In *Proc. 1996 Int. Conf. Knowledge Discovery and Data Mining* (*KDD'96*), Portland, Oregon, USA, Aug., 1996, pp.226–231.

[4] Zhang T, Ramakrishnan R, Livny M. BIRTH: An efficient data clustering method for very large databases. In *Proc. the ACM-SIGMOD Int. Conf. Management of Data*, Montreal, Quebec, Canada, June, 1996, pp.103–114.

[5] Sudipto Guha, Rajeev Rastogi, Kyuseok Shim. CURE: A clustering algorithm for large databases. In *Proc. the ACM SIGMOD Int. Conf. Management of Data*, Seattle, Washington, USA, June, 1998, pp.73–84.

[6] Karypis G, Han E-H, Kumar V. CHAMELEON: A hierarchical clustering algorithm using dynamic modeling. *IEEE Computer*, 1999, 32(8): 68–75.

[7] Sheikholeslami G, Chatterjee S, Zhang A. WaveCluster: A multi-resolution clustering approach for very large spatial databases. In *Proc. 1998 Int. Conf. Very Large Databases,* New York, August, 1998, pp.428–439.

[8] Agrawal R, Gehrke J, Gunopulos D, Raghavan P. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. the 1998 ACM SIGMOD Int. Conf. Management of Data*, Seattle, Washington, USA, June, 1998, pp.94–105.

[9] Jiang M F, Tseng S S, Su C M. Two-phase clustering process for outliers detection. *Pattern Recognition Letters,* 2001, 22(6/7): 691–700.

[10] Venkatesh Ganti, Johannes Gehrke, Raghu Ramakrishnan. CACTUS-clustering categorical data using summaries. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining*, August, 1999, pp.73–83.

[11] UCI Repository of Machine Learning Databases. http://www.ics.uci.edu/~mlearn/MLRRepository.html

[12] Huang Z. A fast clustering algorithm to cluster very large categorical data sets in data mining. In *Proc. SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, Tucson, Arizona, USA, May, 1997, pp.146–151.

[13] David Gibson, Jon Kleiberg, Prabhakar Raghavan. Clustering categorical data: An approach based on dynamic systems. In *Proc. 1998 Int. Conf. Very Large Databases*, New York, August, 1998, pp.311–322.

[14] Zhang Yi, Ada Wai-Chee Fu, Chun Hing Cai, Peng-Ann Heng. Clustering categorical data. In *Proc. 2000 IEEE Int. Conf. Data Engineering*, San Deigo, USA, March, 2000, p.305.

[15] Eui-Hong Han, George Karypis, Vipin Kumar, Bamshad Mobasher. Clustering based on association rule hypergraphs. In *Proc. 1997 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, Tucson, Arizona, USA, May, 1997, pp.78–85.

[16] Wang Ke, Xu Chu, Liu Bing. Clustering transactions using large items. In *Proceedings of the 1999 ACM International Conference on Information and Knowledge Management*, Kansas City, Missouri, USA, November, 1999, pp.483–490.

[17] Sudipto Guha, Nina Mishra, Rajeev Motwani, Liadan O'Callaghan. Clustering dsta streams. In *The 41st Annual Symposium on Foundations of Computer Science*, Redondo Beach, California, USA, November, 2000, pp.359–366.

HE Zengyou received his M.S. degree in computer science from Harbin Institute of Technology (HIT) in 2002. He is currently a Ph.D. candidate in the Department of Computer Science and Engineering, HIT. His main research interests include data mining, multi-database systems and approximate query answering.

XU Xiaofei received his M.S. and Ph.D. degrees in computer science from HIT in 1985 and 1988 respectively. He is currently a professor in the Department of Computer Science and Engineering, HIT. His main research interests include CIMS and database systems.

DENG Shengchun received his Ph.D. degree in computer science from HIT in 2002. He is currently an associate professor in the Department of Computer Science and Engineering, HIT. His main research interests include data mining and data warehouse.