ItCompress: An Iterative Semantic Compression Algorithm

H. V. Jagadish[1] Raymond T. Ng[2] Beng Chin Ooi [3] Anthony K. H. Tung[3,4]

[1]University of Michigan
1301 Beal Ave,
Ann Arbor, MI, 48109-2122
jag@eecs.umich.edu

[2]University of British Columbia
2366 Main Mall,
Vancouver, B.C., V6T 1Z4
rng@cs.ubc.ca

[3]Natl. University of Singapore
3 Science Dr 2,
Singapore 117543
{ooibc, atung}@comp.nus.edu.sg

[4]Contact Author

Abstract

Real datasets are often large enough to necessitate data compression. Traditional 'syntactic' data compression methods treat the table as a large byte string and operate at the byte level. The tradeoff in such cases is usually between the ease of retrieval (the ease with which one can retrieve a single tuple or attribute value without decompressing a much larger unit) and the effectiveness of the compression. In this regard, the use of semantic compression has generated considerable interest and motivated certain recent works.

In this paper, we propose a semantic *compression algorithm called ItCompress* ITerative Compression*, which achieves good compression while permitting access even at attribute level without requiring the decompression of a larger unit. ItCompress iteratively improves the compression ratio of the compressed output during each scan of the table. The amount of compression can be tuned based on the number of iterations. Moreover, the initial iterations provide significant compression, thereby making it a cost-effective compression technique. Extensive experiments were conducted and the results indicate the superiority of ItCompress with respect to previously known tehniques, such as 'SPARTAN' and 'fascicles'.*

1 Introduction

Advances in information technology have necessitated the creation of massive high-dimensional tables required for new applications such as corporate data warehouses, network-traffic monitoring and bio-informatics. The sizes of such tables are often in the range of terabytes, thereby making it a challenge to store them efficiently. In order to reduce the respective sizes of such tables, an obvious solution is the use of traditional data compression methods which are statistical or dictionary-based (e.g., Lempel-Ziv [17]). Such methods are 'syntactic' in nature since they view the table as a large byte string and operate at the byte level.

More recently, compression techniques, which take semantics of the table into consideration during compression [9, 1], have received considerable attention. In general, these algorithms first try to derive a descriptive model, M, of the database by taking into account the semantics of the attributes and then separate them into the following three groups with respect to M:

1. Data values that can be derived from M.
2. Data values essential for deriving the data values in (1) using M.
3. Data values that do not fit M i.e., outliers.

By storing only the model M together with the second and third groups of data values, compression is achieved since M typically takes up substantially less storage space than the original database. Such semantic compression generally has the following advantages over syntactic compression:

- **More Complex Analysis**
  Since the semantics of the data are taken into consideration, complex correlation and data dependency between the data attributes can be exploited in case of semantic compression of data. Note that this is not supported in case of syntactic compression methods since the database is viewed as a large byte string in such methods. Further, the exploratory nature of many data analysis applications implies that exact answers are usually not needed and analysts may prefer a fast approximate answer with an upper bound on the error of approximation. By taking into consideration the error tolerance that is acceptable in each attribute, semantic compression can be used to perform **lossy compression** to enhance the compression ratio. (The benefits can be substantial even when the level of error tolerance is low).

- **Fast Retrieval**
  Given a massive table, only *certain* rows of the table are typically accessed to answer database queries. As such, it is desirable to be able to decompress only certain tuples in the database, while allowing the other tuples to remain uncompressed. Since syntactic compression methods,

such as gzip, are unaware of the record boundary, it is usually not possible to do so without uncompressing the whole database. In fact, it has even been suggested [12, 13] that tables are better compressed column-wise. Separate compression of individual tuples, and even individual attributes, is possible. However, syntactic compression is usually not effective on very small strings. As such, this sort of fine granularity compression is not used frequently.

Semantic compression permits local reconstruction of selected tuples and even attributes without having to reconstruct the entire table. In fact, it is even possible to store the compressed data in a relational database, thereby making the query optimization and indexing techniques of relational databases available also for compressed data.

- **Query Enhancement**
  In addition to the compression itself, there could be intrinsic value in obtaining the descriptive model M for semantic compression. For example, in [1] where M is a set of classification or regression trees, the following rule could have been found "if $X = a$, then $Y = b$ with 100% accuracy". In such a case, for a query searching for tuples with $X = a$ and $Y = c$, an empty answer set could be returned very efficiently. Such side benefits are not available when syntactic compression is used.

Although semantic compression has several advantages over syntactic compression, the two types of compression are *not* mutually exclusive. In fact, it has been shown in [9] that applying semantic compression before syntactic compression results in better compression performance than from either syntactic compression or semantic compression. (However, syntactic compression used in the second phase will nullify the fast retrieval benefit discussed earlier for semantic compression.)

In this paper, we propose a new semantic compression scheme based on the selection of **representative rows**. Each tuple in the table is assigned to one of the representative rows and its attribute values are defaulted to be the same with the assigned representative rows unless the actual value differs from the default value by more than an acceptable error threshold. In such cases, **outlying values** are specifically stored for the row. Our scheme is similar to clustering, with each representative row considered like a cluster representative. However, there is a significant difference due to the outlying values that are possible. These attributes could have values in a row wildly different from its assigned representative row. As such, by most standard metrics, the distance between a row and its representative could be very large. Let us illustrate the concept with an example:

Example 1 Consider the table in Figure 1(a) which contains 5 attributes and 8 tuples. Let the acceptable error threshold for the numeric attributes age, $salary$ and $assets$ be 5, 25,000 and 50,000 respectively, while no errors are allowed for categorical data. We show a selection of representative rows in Figure 1(c) and the compressed table in Figure 1(b). As can be seen, each row in the compressed table is associated with one of the representative rows using a **representative row ID**

age	salary	assets	credit	sex
20	30,000	25,000	poor	male
25	76,000	75,000	good	female
30	90,000	200,000	good	female
40	100,000	175,000	poor	male
50	110,000	250,000	good	female
60	50,000	150,000	good	male
70	35,000	125,000	poor	female
75	15,000	100,000	poor	male

(a) An Example Table

RRid	Bitmap	Outlying Values
2	01011	20, 25,000
1	11011	75,000
1	11111	
1	01100	40, poor, male
1	01111	50
1	01110	60, male
2	11110	female
2	11111	

(b) Table T_c

RRid	age	salary	assets	credit	sex
1	30	90,000	200,000	good	female
2	70	35,000	100,000	poor	male

(c) Representative Rows

Figure 1. Representative Rows and Compressed Table

(RRid). A bitmap is assigned to each row to provide the position for the outlying values. A '1' at the n^{th} bit indicates that its n^{th} attribute value is within an acceptable error tolerance threshold of the n^{th} attribute value for the representative row, while a '0' indicates otherwise. Thus from the bitmap in the first row, we can see that the values for attribute "age" and "assests" in that row are '20' and '25,000' respectively instead of '30' and '200,000' as indicated by its representative row. □

To compress data according to the scheme, the difficult issue is to choose a good set of representative rows. For this purpose, we develop an algorithm called ItCompress (ITerative Compression) which **iteratively improves** the set of chosen representative rows. From one iteration to the next, new representative rows may be selected, and old ones discarded. A key analytical result of this paper is the "convergence" theorem showing that, even though the representative rows may keep changing, each iteration monotonically improves the global quality. In fact, for many cases, the rate of convergence is high i.e., even a small number of iterations may be sufficient to deliver significant compression performance. Furthermore, each iteration of the algorithm requires only a single scan over the data, leading to a fast compression scheme.

The rest of this paper is organized as follows. Section 2 gives a formal definition of the problem of semantic data compression, while our proposed ItCompress algorithm is presented in Section 3. In Section 4, we briefly describe competing approaches, and discuss their comparative merits. In

Section 5, we present results from an extensive experimental study comparing our approach with these approaches. Section 6 describes other related work, while Section 7 concludes with directions for future work.

2 Problem Description

Given a table T, which has m attributes $X_1, ..., X_m$ and n rows, we use $R[X_i]$ to represent the value of A_i for row R. We denote the domain of attribute X_i as $dom(X_i)$. Our aim is to perform a lossy compression on T such that the values reconstructed from the compressed table satisfy certain error tolerances for each column. We denote these error tolerances as a vector $e = [e_1, ..., e_m]$ with e_i being the error tolerance for X_i. The value e_i is interpreted differently depending on the (domain) type of X_i. Our techniques are applicable irrespective of the specific definitions chosen for tolerance. For instance, we could use edit distances for string types, or distance to the closest common ancestor for classification types. To keep matters concrete, in all examples and experiments in this paper, we focus on two of the most popular types: An attribute X_i is said to be **numeric** if the values in $dom(X_i)$ can be ordered while attributes with unordered, discrete domain values are said to be **categorical**. With these, we associate the following tolerance rules:

1. **Categorical**
   For a categorical attribute, the tolerance e_i defines an upper bound on the probability that the approximate value of X_i in T_c is different from the actual value in T. This means that given $X_i = x$ for a particular row in T and $X_i = x'$ for the same row in T_c, $Prob(x = x') \geq 1 - e_i$.

2. **Numeric**
   Given that the value of an attribute X_i is x in T and that x' is it corresponding value in T_c, then the tolerance e_i defines the upper bound that x' can deviate from x. This means that x should be within the range $[x' - e_i, x' + e_i]$.

Given the error tolerance specifications, our aim is to derive a compression scheme that minimizes total storage while satisfying these criteria.

Definition 2.1 Compression Scheme
Given the table T, our basic compression scheme consists of two parts, the set of **representative rows** P and the compressed table T_c

1. **Representative Rows**
   The set of representative rows P consists of a set of k rows $\{P_1, ..., P_k\}$ where each row $P_i \in (dom(X_1)) \times (dom(X_2)) \times ...(dom(X_m))$. We say that a row in R matches a representative row P_i on attribute X_i if one of the following conditions is true:

   (a) $P[X_i] - e_i \leq R[X_i] < P[X_i] + e_i$ if X_i is numeric

   (b) $R[X_i] = P[X_i]$ if X_i is categorical

2. Compressed Table
   For each row R in T, the compressed table T_c has a corresponding row which can be further split into the following three parts:

   (a) A **representative row id**, $RRid$, which indicates that R is most similar to representative row P_{RRid} in the set of representative rows.

   (b) A **bitmap** which has a bit for every attribute. A bit representing X_i is set to 1 if $P_{RRID}[X_i]$ matches $R_{RRiD}[X_i]$ and 0 otherwise.

   (c) An **outlying list** which stores attribute values in R that cannot be inferred satisfactorily from the representative row id and bitmap.

Given our compression scheme, one obvious conclusion is that if we can find a set of k representative rows, $P = \{P_1, ..., P_k\}$ such that all rows are fully matched by at least one member from P, then the number of outlying values that need to be stored will be zero giving rise to very good compression ratio. However, this is not always possible and hence our aim is to minimize the number of outlying values that need to be stored using the notion of **coverage**.

Definition 2.2 Coverage
Let R be a row in T and let P_i be a representative row in P. We say that the coverage of P_i on R, $cov(P_i, R)$ is the number of attributes X_i in which $R[X_i]$ is matched by $P[X_i]$. □

As an example, the coverage of representative row P_1 on the second row of T in Example 1 is 4 since the age, salary, credit, and sex attributes lie within the error tolerance.

Now let us define the total coverage of a set of representative rows P on a table T.

Definition 2.3 Total Coverage
Let P be a set of representative rows $P_1, ..., P_k$ and let the table T contain n rows $R_1, ..., R_n$. For each row, R_i let $P_{max}(R_i)$ be the representative row from P that gives the maximum coverage among P_i ($i \leq i \leq k$) to R_i. We define the total coverage of P on T to be $totalcov(P, T) = \sum_{i=1..n} cov(P_{max}(R_i), R_i)$ □

Maximizing the total coverage is equivalent to minimizing the number of outlying values and thus we have the following problem definition:

Definition 2.4 Maximum Coverage (MC) Problem
Given a table T, an error tolerance vector e and a user-specified value k, find a set of k representative rows P which maximizes $totalcov(P, T)$. □

For ease of reference, we show in Figure 2 the notations used in this paper.

3 The ItCompress Algorithm

The MC problem is easily shown to be NP-hard since a special case is equivalent to the k-center problem [5]. As such,

Notation	Description
e	error tolerance vector
e_i	error tolerance for attribute X_i
$fv(X_j, G(P_i))$	the most frequent value/interval for attribute X_i in group $G(P_i)$
$G(P_i)$	a set of rows which have P_i as the best match
m	number of attributes in a table
n	number of rows in a table
P	a set of representative rows
P_i	the i^{th} representative row in P
$P_i[X_j]$	value of attribute X_i for representative row P_i
$P_{max}(R)$	the representative row that best matches row R
R	a row in a table T
$R[X_j]$	value of attribute X_i for row R
$RRid$	representative row id
T	a table
T_c	a compressed version of T
X	a set of attributes

Figure 2. Notations

our solution to this problem is to find an efficient heuristic. While optimality is not guaranteed for our algorithm, experiments show that it gives a good compression ratio without sacrificing efficiency.

The ItCompress (ITerative COMPRESSion) algorithm is presented in Figure 3. The algorithm begins by picking a random set of k representative rows from the table T. It then iteratively improves this random choice with the objective of increasing the total coverage over the table to be compressed. There are two phases in each iteration:

Phase 1: (Step 3 of ItCompress) In this phase, each row R in T is assigned to a representative row $P_{max}(R)$ that gives the most coverage to R among the members of P. Let $G(P_i)$ denote the set of rows that are assigned to a representative row P_i.

Phase 2: (Steps 4 to 6 of ItCompress) In this phase, a new set of representative rows is computed. Each new P_i is computed by setting each attribute value $P_i[X_j]$ to be $fv(X_j, G(P_i))$ which denotes the most frequently occurring value/interval for attribute X_j in $G(P_i)$.

For a categorical attribute, this can easily be done by keeping count of the number of occurrences for each categorical value in $G(P_i)$ during Phase 1. For a numeric attribute, $fv(X_j, G(P_i))$ is an interval of width $[x - e_j, x + e_j]$, $x \in dom(X_j)$ that is most frequently matched by the rows in $G(P_i)$. An efficient mechanism is to partition the range for X_j into micro-intervals of size that are significantly smaller than e_j say $e_j/10$ and keep track of the frequency of occurrence of each such micro-interval in Phase 1. A sliding window of size $2 * e_j$ is then moved along these sorted micro-intervals to find the range that is most frequently matched. This method ensures a linear time algorithm for computing $fv(X_j, G(P_i))$ while ensuring that the error in estimating $fv(X_j, G(P_i))$ is not more than the size of the micro-interval.

Algorithm ItCompress
Input: A table T, a user specified value k and an error tolerance vector e.
Output: A compressed table T_c and a set of representative rows $P = \{P_1, ..., P_k\}$.

1. Pick a random set of representative rows P
2. While $totalcov(P, T)$ is increasing do
3. { For each row R in T, find $P_{max}(R)$
4. Recompute each P_i in P as follow:
5. { For each attribute X_j,
6. $P_i[X_j] = fv(X_j, G(P_i))$
7. }
8. }

Figure 3. The ItCompress Algorithm

The above two phases are repeated until there is no improvement in $totalcov(P, T)$. In some practical situations, it can also be specified that termination will occur if the improvement in $totalcov(P, T)$ is negligible in comparison to the previous iteration. Example 2 demonstrates the running of the ItCompress algorithm.

Example 2 Consider again the table in Figure 1 and let us assume that the error tolerance vector e is {5, 25000, 50000, 0, 0 }. Assuming that $k = 2$ and that the first and second row of the table are picked as the initial representative rows, we depict the situation for the first iteration of ItCompress in Figure 3. The representative rows P_1 and P_2 are shown in Figure 4(a) while Figure 4(b) and 4(c) show the rows that are assigned to P_1 and P_2 respectively. We leave it to readers to verify that each row is assigned to the representative row that best matches it (arbitrarily breaking ties). Having done so, P_1 and P_2 are recomputed by assigning to each attribute the most frequently occurring value/interval (highlighted in bold).

This new set of representative rows is shown in Figure 5(a) and we highlight the changes from the previous set of representative rows in bold. With this change, the rows in the table are again reassigned and only one of the row changes membership from $G(P_2)$ to $G(P_1)$. Note that the total coverage of the two representative rows improves from 25 to 31 as we move through the two iterations. The iterative process continues until there is no improvement in $totalcov(P, T)$. □

Note that throughout the ItCompress algorithm, the error bound for categorical attribute is not utilized as part of the optimization. However, this can be easily done at the end of the algorithm. Given $G(P_i)$, the set of rows which have P_i as the best match, we can compute for each categorical attribute X_j, the frequency of occurrence of $P_i[X_j]$ within $G(P_i)$. Let us denote this frequency of occurrence as $freq(P_i[X_j], G(P_i))$. Since we have an error tolerance of e_j for X_j, we will remove up to $e_j/(1 - e_j) \times freq(P_i[X_j], G(P_i))$ outlying values from the rows in $G(P_i)$ as long as these outlying values are in the domain of X_j. These removed outlying values can be assumed to be $P_i[X_j]$ without invalidating the error-bound. From here, we can see that ItCompress is indirectly utilizing the error tolerance for categorical attributes by trying to maximize coverage and thus allowing more outlying values to be removed.

RRid	age	salary	assets	credit	sex
1	20	30,000	25,000	poor	male
2	25	76,000	75,000	good	female

(a) Representative Rows

age	salary	assets	credit	sex
20	**30,000**	25,000	**poor**	**male**
60	**50,000**	**150,000**	good	**male**
70	**35,000**	**125,000**	**poor**	female
75	**15,000**	**100,000**	**poor**	**male**

(b) $G(P_1)$:Rows assign to P_1

age	salary	assets	credit	sex
25	**76,000**	75,000	**good**	**female**
30	**90,000**	**200,000**	**good**	**female**
40	**100,000**	**175,000**	poor	male
50	110,000	**250,000**	**good**	**female**

(c) $G(P_2)$:Rows assign to P_2

Figure 4. Iteration 1 for Example 2

RRid	age	salary	assets	credit	sex
1	**70**	30,000	**125,000**	poor	male
2	25	**90,000**	**175,000**	good	female

(a) Representative Rows

age	salary	assets	credit	sex
20	30,000	25,000	poor	male
40	100,000	175,000	poor	male
60	50,000	150,000	good	male
70	35,000	125,000	poor	female
75	15,000	100,000	poor	male

(b) $G(P_1)$:Rows assign to P_1

age	salary	assets	credit	sex
25	76,000	75,000	good	female
30	90,000	200,000	good	female
50	110,000	250,000	good	female

(c) $G(P_2)$:Rows assign to P_2

Figure 5. Iteration 2 for Example 2

3.1 Convergence and Complexity

Given the ItCompress algorithm described above, we have the following theorem:

Theorem 3.1 The total coverage of P on T is non-decreasing for every iteration in ItCompress.

Proof: Our aim is to show that the $totalcov(P,T)$ either increases or remain the same in both Phases 1 and 2.
In Phase 1, this is trivial since each row is assigned a representative row P_i that provide the most coverage. If $P_{max}(R_i)$ changed for a row R_i, then it means that
$cov(R_i, P_{max}(R_i))$ has increased, otherwise without a change in $P_{max}(R_i)$,$cov(R_i, P_{max}(R_i))$ would have remained the same. Since all rows either have the same or increased coverage for the same set of P, $totalcov(P,T)$ must have increased or remained the same.
In Phase 2, we first observe that $\sum_{R\in G(P_i)} cov(R,P_i)$ is equal to $\sum_{j=1..m} match(G(P_i),X_j)$ where $match(G(P_i),X_j)$ is the number of rows in $G(P_i)$ that match P_i on attribute X_j. Since $fv(X_j,G(P_i))$ is chosen such that the number of rows from $G(P_i)$ that match $P_i[X_j]$ is maximum, we are also maximizing $\sum_{R\in G(P_i)} cov(R,P_i)$. Since this is done for each pattern $P_i \in P$, we are thus increasing or maintaining $totalcov(P,T)$.

As can be seen, both Phase 1 and 2 either increase or maintain $totalcov(P,T)$, thus proving the theorem. □

From Theorem 3.1, we can conclude that ItCompress will eventually converge since $totalcov(P,T)$ is finite.

We now look at the issue of efficiency. Since ItCompress iteratively goes through the n rows in the table and matches each of them against the k representative rows, the number of rows compared is kn. Each row comparison requires $2m$ operations, where m is the number of columns. Thus, the run-time complexity for Phase 1 is $O(kmnl)$, where l is the number of iterations. In Phase 2, computing each new P_i requires going through all the domain values/intervals of each attribute. Assuming that the total number of domain values/intervals is d, then Phase 2 will have a run time complexity of $O(kdl)$. Thus, the total run time complexity of ItCompress is $O(kmnl + kdl)$. Since k, d, m and l are usually much less than n, we infer that ItCompress has a linear running time of $O(n)$.

To further reduce the running time in practice, we run ItCompress on a sample drawn from a large table and find a set of representative rows P for the sample. The remaining remaining rows in the table are then assigned to the best matched member in P. Experiments in the next section will show that a 5% to 10% sample is sufficient to produce good compression in this manner.

4 Discussion

Two semantic compression algorithms [9, 1] have previously been suggested in the literature. Both these algorithms are quite complex, and the ItCompress algorithm described above appears to be much less sophisicated. While an extensive performance comparison will be presented in the next section, here we will highlight some of the differences and motivate some of the design choices made in ItCompress.

4.1 Previously Known Semantic Compression Algorithms

In this section, we present a brief sketch of the two previously known semantic compression algorithms that we must compare ourselves against.

The fascicles algorithm presented in [9] is the first semantic compression algorithm developed for tables and relations. Given a table of m columns and a user-specified value of u

($u \leq m$), the algorithm extracts a model M consisting of w fascicles, each of which is represented by a u-tuple. The u columns are called compact attributes because these are columns with very similar values (i.e., values within the error tolerance) for all the rows assigned to the fascicle.

While the fascicles algorithm determines the u compact columns locally on a per fascicle basis, SPARTAN [1] tries to separate the m columns into a set of predictor attributes and a set of predicted attributes globally for the entire relation. The model M, in this case, is simply the set of the predictor attributes. SPARTAN identifies the predictor columns by constructing Bayesian Network and CaRTs (classification and regression trees).

As seen in the fascicle algorithm and SPARTAN, the key aspects that differentiate one semantic compression algorithm from another are the exact definition of the model M used to compress the database and how it is constructed.

4.2 Simplicity and Directness

In both fascicles and SPARTAN, the process of compression can generally be separated into 2 steps.

Step 1: Finding a set of patterns or rules.
Step 2: Using the discovered patterns/rules in Step 1, form the **global model,** M for compressing the database.

To assess the usefulness of the patterns/rules in Step 1, criteria like length of the patterns or accuracy of the rules are used to guide the mining algorithms. There are however no direct guarantee that the patterns/rules discovered using such criteria are actually useful in forming a good global model for compression. For example consider an example in which the following 6 rules are found by SPARTAN:

- Rule 1: $X_1, X_2, X_3 \rightarrow X_7(100\%)$
- Rule 2: $X_4, X_5, X_6 \rightarrow X_8(100\%)$
- Rule 3: $X_3, X_4, X_5 \rightarrow X_7(80\%)$
- Rule 4: $X_3, X_4, X_5 \rightarrow X_8(80\%)$
- Rule 5: $X_1 \rightarrow X_2(80\%)$
- Rule 6: $X_1 \rightarrow X_6(80\%)$

In this example, although both Rule 1 and 2 have 100% prediction accuracy but utilizing them in the global compression model will require the storage of 6 predictor attributes (i.e. X_1,..., X_6). On the other hand, Rule 3 to 6 allow us to store only 4 predictor attributes (i.e. X_1, X_3, X_4 and X_5) although more outliers are expected due to less accurate rules. The tradeoff in performance between these two choices is not entirely clear.

Based on this example, we can seen that it is entirely possible for SPARTAN to find excessive number of patterns/rules that are not useful in constructing a good global compression model while other potentially useful rules can be missed based on the criteria adopted in Step 1. This conclusion seems unsatisfactory considering that we much go through the complex process of training Bayesian Network in SPARTAN (which have a time complexity of $O(m^4 * n)$ [1], m being the number of columns and n the number of randomly sampled rows). Likewise, fascicles suffers from similar difficulties since it too has a two-stage process.

Furthermore, since the selection of the optimal set of patterns/rules in the second step is NP-hard for both fascicles and SPARTAN, greedy algorithms are used for this purpose, with no guarantees on the quality of results obtained.

ItCompress on the other hand adopt a simple philosophy of "direct optimization". Since the aim of a compression algorithm is to reduce the storage requirement for a database, ItCompress directly use this as a optimization criteria and ensure that only patterns which improve the compression are found in each step of its iterations [1] . While the optimization problem is NP-hard in our case as well, the heuristic used is an iterative hill-climbing technique that can recognize when it has reached a (local) maximum. As mentioned earlier, this simple approach result in a much lower time complexity compared to SPARTAN.

4.3 Constraining the Optimization

If one views the compression task as an optimization problem, a question to ask is what constraints are imposed on feasible solutions by virtue of the choice of solution technique.

SPARTAN imposes a constraint that each attribute must be either a predicted or predictor attribute **globally**. In consequence, SPARTAN is not able to exploit situations where there is only **local** "column-wise" dependency in a dataset (i.e., exhibited by a subset of the rows). For example, "age<20" could be a strong predictor for "assets<50,000". However "age>20" might give no good indication on a person's assets. In such a case, age is said to give only a "local" prediction for assets and hence will not be suitable as a predictor attribute since SPARTAN is constraining its compression model to use only global dependency exhibited by all the rows.

Similarly, a fascicle has the constraint that there must be u compact attributes which are matched **completely** by every tuple in it. If even one tuple has a different value for one of the u attributes, it cannot be retained in the fascicle.

ItCompress overcomes this problem by grouping rows based on approximate matching to the representative row and trying to maximize the number of matching columns without any constraints.

4.4 Tuning Parameters

Every algorithm has engineering parameters that must be specified for effective performance.

For instance, consider u, the number of compact attributes required in a fascicle. For a fascicle with n rows, the saving in term of storage will be in the order of $O(un)$. Because of this, the result of the compression can be rather sensitive to u. A small value of u will obviously give little compression even when n is large. A large value of u will result in only a

[1] This philosophy is inspired by the proposal to have a microeconomic view on data mining in [11]

few rows in each fascile, making the value of the this product small. There is no easy way to determine an optimum choice for u.

Similarly, the parameters for the construction of Bayesian Network and CARTs which is needed by SPARTAN are also not easily selected. The is especially true for Bayesian Network since the network structure must first be inferred before any training can take place.

ItCompress, too has user-tunable parameters. However, these parameters can be tuned more easily due to the simplicity of the algorithm. One is the the number of representative rows. Too large a number will result in little compression. Too small a number may leave too many attribute values stored as exception, again leading to little compression. Fortunately, our experimental results show that there is a rather flat optimum region, and the quality of compression is not greatly affected by the choice of value for this parameter.

Another parameter in ItCompress is the number of iterations it is run for. Here, we know when to stop, since the improvement is easily measured from one iteration to the next. As such, it is not necessary to have an a priori determination of this parameter value.

5 Performance Study

In this section, we study the performance of ItCompress, and evaluate its senstivity to various parameters. We also compare its performance, not only against the fascicles and SPARTAN algorithm, but also against gzip [17, 18], a well-known syntactic compression algorithm.

Since the SPARTAN system is proprietary and the code is non-trivial for us to reproduce from scratch, we decided to follow the experiments reported in [1] and compare our results against those described in it. We implemented and ran the other three algorithms. Faithful reproduction of datasets provides a fair basis to compare the compression obtained by the four algorithms. To compare running times, we performed our experiments on a single processor PC with a 700Mhz AMD Duron processor and 256MB of main memory, which is clearly less superior to the system used in [1] that has four Pentium 700Mhz processors and 1 GB of memory. This provides an advantage to SPARTAN over the other algorithms, and ItCompress in particular.

Syntactic compression can optionally be applied after semantic compression, since the two techniques are complementary. As such, in addition to running gzip by itself, we also ran gzip on the results of the other algorithms. For ItCompress and fascicles, we are able to report results both with and without this final syntactic compression.

In the case of SPARTAN, the results in [1] are obtained after applying gzip on top of the initial compression by fascicles and SPARTAN [2]. As such, we do not have the numbers for compression by SPARTAN alone, without the gzip that follows.

[2]For fascicle, gzip is applied on the whole compressed dataset while for SPARTAN, gzip is applied on the predictor attributes.

Real-life Data Sets. As in the experiments in the SPARTAN paper[1], we select the following three real-life datasets for experiments.

- *Corel.(http://kdd.ics.uci.edu/databases/CorelFeatures)*
  This dataset consists of 32 numerical attributes and contains 68,040 tuples. Since only a 10.5MB subset is used out of the 20MB dataset in [1], we followed likewise and randomly selected twenty such subsets in order to reduce the error due to variations in the sample. Experiments show that the difference in performance over the twenty subsets is negligible. Each subset consists of around 35,000 tuples.
- *Forest-cover. (http://kdd.ics.uci.edu/databases/covertype)*
  This dataset was also downloaded from the given website. It is a 75.2MB dataset containing 581,000 tuples with 10 numeric and 44 categorical attributes describing the elevation, slope, soil type etc for a forest cover.
- *Census.* (www.bls.census.gov)
  This dataset is obtained directly obtained from the authors of [1]. It consists of 7 categorical attributes and 7 numeric attributes selected from a census dataset. There are 676,000 tuples in the dataset and it takes up a storage of 28.6 MB.

Default Parameter Settings. For SPARTAN, we used the default parameter values used in [1]. To obtain the result for fascicles without gzip, we also used the optimal settings provided in [1]. For ItCompress, the default value for error tolerance was fixed at 0 for all categorical attributes. For a numeric attribute X_i, we specified e_i as a percentage of the width of the range of X_i values in the table, and we had set this percentage to 1% by default. For other parameters, the default value of k was fixed at 300 and the sampling rate was 10%. The number of iterations for ItCompress was limited to 3, unless otherwise stated.

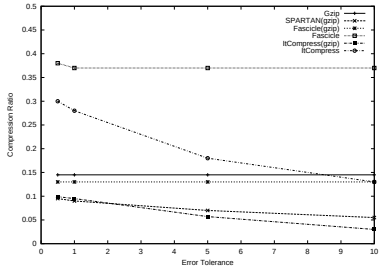
5.1 Effect of Random Initialization

The initialization of ItCompress is based on a random selection of representative rows. We investigated the variation in compression performance produced by ItCompress due to this random selection. For this purpose, we performed 5 sets of experiments using a different set of initial representative rows each time. To see if this random choice becomes more significant when more or fewer representatives are chosen, we varied k, and again repeated the experiment five times with different random initial representative rows for each value of k. For the same reason, we also repeated the experiment with different datasets. Our results are shown in Figure 6 where each column of the table represents one set of five repetitions for a chosen dataset and specified value of k. As can be seen, all values in any column are almost identical, indicating that the variance in compression ratio due to different random initializations is insignificant for all the three datasets with the value of k ranging from 50 to 500. We thus have reason to believe that the compression ratio of ItCompress is stable despite the random initialization. To be doubly sure, all our readings for ItCompress in the next two sections are reported as the average over 5 runs. Note that we did not observe a significant variance between runs in any of these cases.

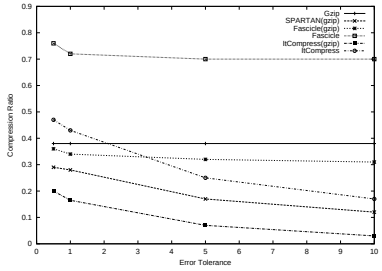
	Forest-Cover			Corel			Census		
Exp. No.	$k = 50$	$k = 300$	$k = 500$	$k = 50$	$k = 300$	$k = 500$	$k = 50$	$k = 300$	$k = 500$
1	0.451	0.283	0.271	0.538	0.431	0.421	0.505	0.281	0.270
2	0.455	0.281	0.270	0.529	0.430	0.419	0.492	0.277	0.268
3	0.447	0.279	0.270	0.528	0.430	0.420	0.501	0.284	0.270
4	0.443	0.281	0.269	0.533	0.431	0.420	0.503	0.275	0.272
5	0.447	0.285	0.270	0.535	0.430	0.421	0.497	0.281	0.272

Figure 6. Compression Ratio Under Random Initialization

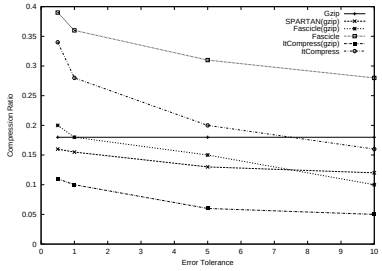
5.2 Empirical Comparison of Algorithms



(a) Forest Cover



(b) Corel



(c) Census

Figure 7. Error Threshold vs Compression Ratio.

We next compare ItCompress with the other compression algorithms in terms of both effectiveness and efficiency. In Figure 7, we vary the error tolerance threshold from its default value and look at the compression ratio achieved by the various algorithms on the three datasets. From the graphs in Figure 7, we make the following observations:

- Combining syntactic and semantic compression generally gives better performance than pure syntactic or semantic compression. The only exceptional case is noted for the Corel dataset in which ItCompress by itself outperforms fascicle(gzip) when the error tolerance threshold is high.
- For pure semantic compression, the compressed tables produced by ItCompress are around 1.2 to 1.7 times smaller than those produced by fascicles for the smallest error tolerance threshold (i.e. 0.05%) while the difference increases to 2 to 3 times for larger thresholds. ItCompress exhibits a much steeper improvement in compression ratio as the error tolerance threshold is increased as compared to fascicles. This result clearly shows that ItCompress, which dynamically switches the membership of a row to a group that best matches it, is able to give better compression than fascicle which fixes the membership of row once it is assigned to a particular group.
- For "combined" algorithms involving both semantic compression and gzip, ItCompress(gzip) always yields better compression except on the Forest Cover dataset where SPARTAN(gzip) is only slightly better for small error tolerance threshold. For other datasets, however, ItCompress(gzip) is always the clear winner producing compressed tables that are 1.5 to 3 times smaller than the closest competitor. This again illustrates that the underlying strategy of ItCompress, which simply aims to maximize the total coverage of its representative rows, is a more effective method than other semantic compression algorithms.

Given the impressive compression ratio that is achieved by ItCompress, the natural question to ask is whether this is done by paying a price in term of efficiency.

To answer this question, we compare the running time of ItCompress, fascicles and SPARTAN in Figure 8 for an error tolerance of 1%. We expect error tolerance to have insignificant impact on the running time of the three algorithms and we pick an error tolerance of 1% since the running time for SPARTAN is taken from [1] which uses the same error tolerance threshold. Note that for SPARTAN, running time is dependent on the CaRT selection algorithm that is being used. There are three such algorithms, **Greedy**, **WMIS(Parent)** and **WMIS(Markov)**. For each dataset, we take SPARTAN 's running time to be the minimum one among the three algorithms. As can be seen from the table, fascicles always has

Data Set	Running Time(sec)		
	ItCompress	Fascicles	SPARTAN
Forest-Cover	169.66	151.00	670.00
Corel	23.29	20.12	80.73
Census	57.93	50.51	153.00

Figure 8. Comparison of Running Time

the best running time among the three algorithms. Compared to fascicles, ItCompress 's running time is around 10% to 15% more. We feel that this small increase in running time is justifiable compared to the large gain in compression ratio. Moreover, one can always trade off the compression ratio slightly by running fewer iterations of ItCompress to substantially reduce the running time. Please see Figures 9(c) and 10(c). For SPARTAN, its running time is almost 2.5 to 4 times more than ItCompress. This clearly indicates that ItCompress can achieve comparable compression ratio with SPARTAN while maintaining a significant advantage in terms of running time.

5.3 Effect of Parameter Settings

Having compared ItCompress to the other compression algorithms, we will next investigate the effect that different parameter settings have on ItCompress. To keep the number of parameter setting combinations small, we will only vary the setting for one parameter at a time, while keeping the setting for other parameters to their default values. Experiments are conducted on all three real-life datasets.

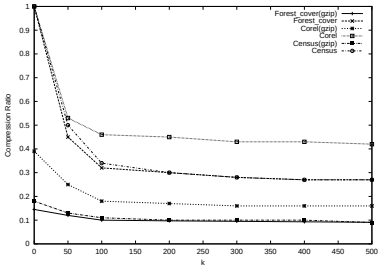
5.3.1 Effect of Parameter Settings on Compression Ratio

We will first look at how the setting of parameters affects the compression ratio of ItCompress.

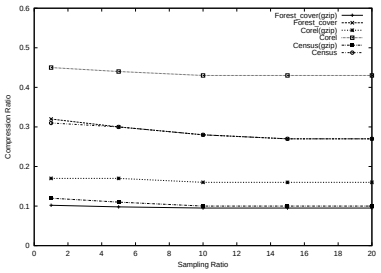
Varying number of representative rows. In Figure 9(a), we vary k, the number of representative rows, from 0 to 500 and look at the compression ratio achieved by ItCompress and ItCompress(gzip) on the three datasets. From the graph, we can see that increasing k will improve the compression ratio of ItCompress and ItCompress(gzip), but the improvement is only marginal from $k = 100$ onwards. This improvement is due to the fact that with higher value of k, each row is more likely to be allocated to a representative row that could match it on higher number of attributes. This results in fewer outlying values and reduces the total storage needed for them.

However, we also note that the storage for representative rows will increase with k. Thus if k is subsequently increased to an extreme value where the reduction in outlying values is not enough to offset the additional storage need for the representative rows, then the compression ratio will increase instead.

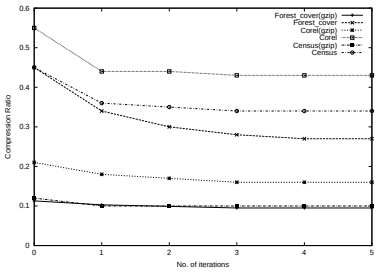
Although this situation does not occur easily as evident from our experiments, we will still try to start with a small value for k when we are trying to optimize the performance of ItCompress. This can then be increased gradually to a point where the increase in compression ratio is negligible. As we had observed earlier, this is not too difficult since there is a large range of values for k where there is no significant increase or decrease in the compression ratio.



(a) Varying number of representative rows



(b) Varying Sampling Ratio

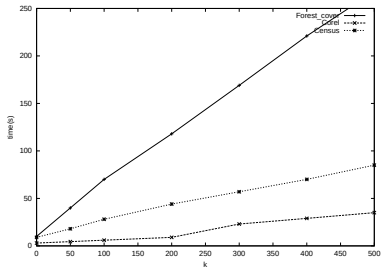


(c) Varying Number of Iterations

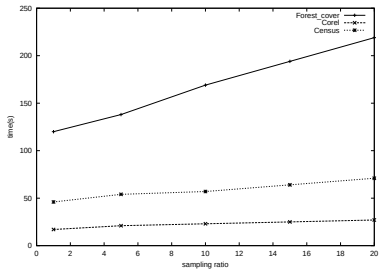
Figure 9. Parameter Settings vs Compression Ratio.

Varying Sampling Ratio. We next vary the sampling ratio from 1% to 20% to see its effect on ItCompress's performance. From Figure 9(b), we observe that while increasing the sampling ratio improves the compression ratio for both ItCompress as well as ItCompress(gzip), this improvement is negligible. Among the three datasets, ItCompress seems to be most affected by the sampling ratio when it is run on the Forest Cover dataset. This is due to the higher dimensionality of the dataset which means that a higher sampling rate is needed to capture the distribution of the data.

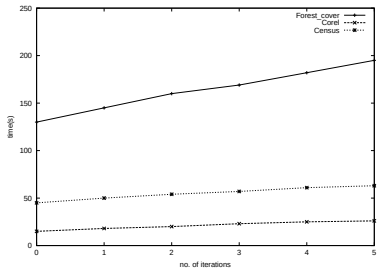
Varying Number of Iterations. Finally, we investigate how the compression ratio of ItCompress improves with the number of iterations. In Figure 9(c), we plot the compression ratio



(a) Varying number of representative rows



(b) Varying Sampling Ratio



(c) Varying Number of Iterations

Figure 10. Parameter Settings vs Running Time.

for ItCompress and ItCompress(gzip) against the number of rounds that ItCompress is allowed to iterate. Note that we consider the number of iterations to be 0 when we randomly pick k representatives and assign the remaining rows to the best matched representative. From the graph, we see that the first 1 to 3 iterations of ItCompress bring about the most improvement in compression ratio, while the performance levels off for higher number of iterations. Among the three datasets, the Forest Cover dataset takes the most number of iterations for the performance to level off. Once again, this is due to its high dimensionality which gives more room for performance improvement in terms of the number of matched attribute values which a database row shares with the closest representative rows.

From this, we can conclude that ItCompress does not need too many iterations in order to achieve good compression. This is important since many scans through the table will result in loss of efficiency.

5.3.2 Effect of Parameter Settings on Running Time

Having seen how the parameter settings affect the compression ratio of ItCompress, we will examine the effect of parameter settings on the running time of ItCompress. Note that we leave out ItCompress(gzip) in this section since the gzip algorithm takes up a negligible amount of running time as compared to ItCompress and thus we will see no difference between the running time of ItCompress and ItCompress(gzip) on the same dataset. Also all running times in this section are obtained from the same set of experiments as in the previous section. Thus, readers should feel free to compare the tradeoff in compression ratio and running time by performing a direct mapping between the two sets of graphs.

Varying k. Figure 10(a) depicts the running time of ItCompress against k. The graph confirms our analysis in the earlier section that the running time of ItCompress is linear with respect to k. By comparing this graph with Figure 9(a), we see that the running time of ItCompress can, in fact, be improved by selecting k to be 100 without sacrificing too much on compression ratio. This also means that the running time we have shown in Figure 8 can be improved without any significant effect on the compression ratio.

Varying Sampling Ratio. As shown in Figure 10(b), the running time of ItCompress is also linear with respect to the sampling ratio. Note that the running time does not tend to zero together with the sampling ratio since our running time is based on the total time that ItCompress takes to run on the sample and the time taken to allocate each row in the table to the best matched representative row at the end of the run. In fact, we can see from the graph that the total running time is dominated mainly by the time taken to allocate each row in the table to the best matched representative at the end while the time spent on discovering representative rows from the sample is generally less significant.

Varying Number of Iterations. From Figure 10(c), we can see that the running time of ItCompress is also linear with respect to the number of iterations. This is consistent with our analysis in Section 3. By comparing Figure 10(c) with Figure 9(c), we can see that the time spent on the 3rd iteration and beyond will *not* bring significant improvement in compression ratio, thus justifying our default value for the number of iterations that we allow ItCompress to run through.

5.4 Performance on Noisy Datasets

In all previous experiments, our tests are performed on the original datasets. One aspect of real-life datasets however is the existence of random noise which might result in a degradation of performance for all the semantic compression algorithms. In this section, we will look at how ItCompress is affected by the amount of random noise that is presented in

Algorithm Corrupt
Input: A table T, a user specified corrupt level, $clevel\%$. **Output:** A corrupted table T' with a noise level of $clevel\%$.

1. For each row R in T,
2. For each attribute X_i,
3. If $rand() \leq clevel\%$
4. corrupt $R[X_i]$
5. Output corrupt row R' to T'

Figure 11. Algorithm for Dataset Corruption

the datasets [3]. Figure 11 illustrates the algorithm for corrupting the datasets. The algorithm essentially goes through every row in the table and attempts to corrupt each in turn. Each attribute in a row have a (corruption level) $clevel\%$ probability of being changed to a random value within the domain of the attribute.

We ran ItCompress on the datasets with corruption level of 5% to 50% and compare it against the fascicle algorithm. Default values are used for the parameter settings of ItCompress while the parameters for fascicles are tuned to give the best performance. As shown in Figure 12, the compressed file size for ItCompress increases linearly with the corruption level for all datasets while the performance of the fascicle algorithm degrades sharply even with a small corruption level (though it levels off afterwards).

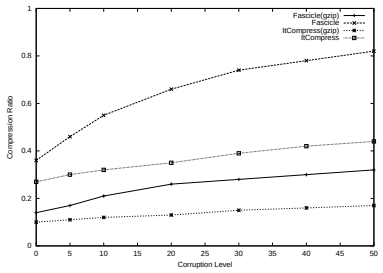
This behavior can be explained by our earlier observation in Section 4 that the compression scheme adopted by ItCompress is less restrictive than that of fascicles algorithm which requires a perfect match for all the compact attributes for the rows in a fascicle. In the presence of noise, a perfect block of fascicle can break down rapidly especially when the number of compact attributes is large as in the case of the Forest Cover and Census datasets. This is because the probability of having a perfect match in the presence of noise drops exponentially with the number of compact attributes considered. As evidenced from Figure 12(a) and 12(c), a large number of compact attributes which orginally bring better compression performance now become a liability to the fascicle algorithm when noises are introduced.

While we are unable to perform any comparison to SPARTAN due to reasons mentioned earlier, we note that SPARTAN makes use of rules that are of the form "$X \rightarrow Y$" in order to perform its compression. Such rules also require perfect match for all the attributes involved in both their L.H.S. and R.H.S. As such, we have every reason to believe that the performance of SPARTAN will also degrade substantially like the fascicle algorithm when noises are introduced.

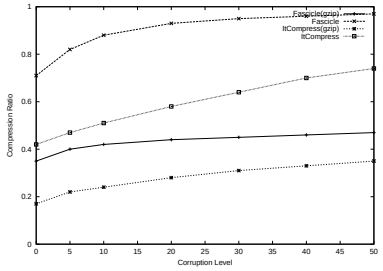
6 More Related Work

Since we have compared ItCompress to both fascicles and SPARTAN in earlier sections, we have only one additional issue to raise in term of decompression efficiency. Since ItCompress essentially finds a best match representative for every row, uncompressing a row is relatively simple and is

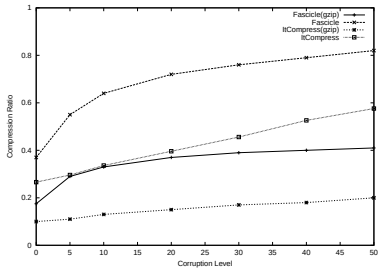
[3] While the original datasets might already contain noises, this is not measurable from our perspective.



(a) Forest Cover



(b) Corel



(c) Census

Figure 12. Noise Level vs Compression Ratio.

performed by first extracting the particular best matched representative row and then substituting the outlying attribute values from the outlying list. The computational complexity of such an operation is $O(m)$. Datasets compressed by SPARTAN do not enjoy this advantage. Since each predicted attribute value is only available after passing the predictor values through the classification and regression trees, a row which has m attributes could take $O(m^2)$ time to be uncompressed in the worst case.

As with other semantic compression algorithms, ItCompress can provide lossy compression within the specified error tolerance. The rearrangement of columns in a relation has been shown to affect the compression achieved in previous studies [13, 12]. Under the proposed scheme, this idea is carried further since the outlying attributes are determined on a per tuple basis with respect to its representative row.

To achieve scalable spatial clustering, it is common practice to reduce the size of the dataset by grouping points into

micro-clusters [16, 6, 15, 14] and then perform the actual clustering on the compressed dataset. However, no error tolerance threshold is allowed along each dimension and the technique is applicable only to numerical attributes. Since ItCompress adopts an iterative refinement approach for compression, in some sense it can be compared to the k-means algorithm [7]. We note however that the k-means algorithm is also not applicable here since it does not give any guaranteed error bounds and converges only on Euclidean distance. Our contribution lies in the development of a semantic compression scheme in which the iterative refinement technique(with convergence) is applicable and in which error tolerance is observed for datasets involving both categorical as well as numerical attributes.

An important aspect of ItCompress is dimensionality reduction (which leads to compression). "Feature reduction" is a fundamental problem in machine learning and pattern recognition. There are many well-known statistical techniques for dimensionality reduction, including Singular Value Decomposition (SVD) [4] and Projection Pursuit [3]. The key difference here is that dimensionality reduction is applied to the entire dataset. In contrast, ItCompress handles the additional complexity of finding different subsets of the data, all of which may permit a reduction on different subsets of dimensions. The same comment extends to FastMap [2], the SVDD technique [10], and the DataSphere technique [8].

7 Conclusion

In this paper, we have presented a simple and general semantic compression algorithm, ItCompress, and demonstrated it to be remarkably effective over a variety of datasets. ItCompress works by choosing random representative rows and then improving upon this choice iteratively. We thus have the possibility of trading off the running time of the compression algorithm for compression ratio. The good news is that the asymptotic compression limit is approached after only a few iterations.

Like other semantic compression algorithms, ItCompress is complementary with respect to byte-oriented syntactic compression. Syntactic compression techniques may be applied to the results of ItCompress to obtain additional compression.

As part of our future work, we are currently looking at how the compressed database and semantic model of ItCompress can be used to speed up more complex mining tasks such as Bayesian Network building or CaRTs construction. Achieving these could mean that more complex semantic models can be discovered efficiently and such models could provide even better semantic compression. ItCompress can thus be used as a "bootstrap" compression algorithm for entering into a positive loop where compression enhances mining and mining, in turn, enhances compression. This is a different form of iterative compression.

References

[1] S. Babu, M. N. Garofalakis, and R. Rastogi. Spartan: A model-based semantic compression system for massive data tables. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, USA, May 2001.

[2] C. Faloutsos and K. Lin. FastMap: a Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. *Proc. 1995 ACM-SIGMOD*, pp. 163–174.

[3] J.H. Friedman and J.W. Tukey. A Projection Pursuit Algorithm for Exploratory Data Analysis. *IEEE Transactions on Computers*, 23, 9, pp 881–889, 1974.

[4] K. Fukunaga. Introduction to Statistical Pattern Recognition. *Academic Press*, 1990.

[5] M. Garey and D. Johnson. *Computers and Intractability: a Guide to The Theory of NP-Completeness*. Freeman and Company, New York, 1979.

[6] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'98)*, pages 73–84, Seattle, WA, June 1998.

[7] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[8] T. Johnson and T. Dasu. Comparing massive high-dimensional data sets. *Proc. 1998 KDD*, pp 229–233.

[9] H. V. Jagadish, J. Madar, and R. Ng. Semantic compression and pattern extraction with fascicles. In *Proc. 1999 Int. Conf. Very Large Data Bases (VLDB'99)*, pages 186–197, Edinburgh, UK, Sept. 1999.

[10] F. Korn, H.V. Jagadish, C. Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. *Proc. 1997 ACM-SIGMOD*, pp. 289–300.

[11] J. M. Kleinberg, C. Papadimitriou, and P. Raghavan. A microeconomic view of data mining. *Data Mining and Knowledge Discovery*, 2:311–324, 1998.

[12] W. K. Ng and C. V. Ravishankar. Block-Oriented Compression Techniques for Relational Databases. TKDE, April 1997, pp 314–328.

[13] F. Olken and D. Rotem. Rearranging Data to Maximize the Efficiency of Compression. *Proc. 1986 PODS*, pp 78-90.

[14] A. K. H. Tung, J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-based clustering in large databases. In *Proc. 2001 Int. Conf. Database Theory (ICDT'01)*, London, U.K., Jan. 2001.

[15] A. K. H. Tung, J. Hou, and J. Han. Spatial clustering in the presence of obstacles. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, Heidelberg, Germany, April 2001.

[16] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: an efficient data clustering method for very large databases. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'96)*, pages 103–114, Montreal, Canada, June 1996.

[17] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 24:550–536, 1978.