Paper Number 510

Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences

Flip Korn[*]
Dept. of Computer Science
University of Maryland
College Park, MD 20742
`flip@cs.umd.edu`

H. V. Jagadish
AT&T Laboratories
Murray Hill, NJ 07974
`jag@research.att.com`

Christos Faloutsos[†]
Dept. of Computer Science and
Inst. for Systems Research
University of Maryland
College Park, MD 20742
`christos@cs.umd.edu`

Abstract

Ad hoc querying is difficult on very large datasets, since it is usually not possible to have the entire dataset on disk. While compression can be used to decrease the size of the dataset, compressed data is notoriously difficult to index or access.

In this paper we consider a very large dataset comprising multiple distinct time sequences. Each point in the sequence is a numerical value. We show how to compress such a dataset into a format that supports ad hoc querying, provided that a small error can be tolerated when the data is uncompressed. Experiments on large, real world datasets (AT&T customer calling patterns) show that the proposed method achieves an average of less than 5% error in any data value after compressing to a mere 2.5% of the original space (*i.e.*, a 40:1 compression ratio), with these numbers not very sensitive to dataset size. Experiments on aggregate queries achieved a 0.5% reconstruction error with under 2% space requirement.

1 Introduction

The bulk of the data in most data warehouses has a time component (*e.g.*, sales per week, transactions per minute, phone calls per day, *etc.*). More formally, these datasets are of N time sequences, each of duration M, organized in an $N \times M$ matrix (N row vectors of dimensionality M). In such databases, decision support (*i.e.*, statistical analysis) requires the ability to perform ad hoc queries. What one would like is a way to compress data in such a way that ad hoc queries are still supported efficiently. In this paper, we introduce a way to do this, for numerical (time sequence) data, at the cost of a small loss in numerical accuracy.

When the dataset is very large, accessing specific data values is a difficult problem. For instance, if the data is on tape, such access is next to impossible. When the data is all on disk, the cost of disk storage, even with today's falling disk prices, is typically a major concern, and anything one can do to

---

[*] Work performed while visiting AT&T.

[†] Work performed while visiting AT&T. Partially supported by NSF grants EEC-94-02384, IRI-9205273, IRI-9625428.

decrease the amount of disk storage required is of value. We, the authors, ourselves have experience with more than one dataset that ran into hundreds of gigabytes, making storage of the data on disk prohibitively expensive. Unfortunately, most data compression techniques require large blocks of data to be effective, so that random access to arbitrary pieces of the data is no longer conveniently possible, making it difficult to issue ad hoc queries, and therefore do not support the sort of random ad hoc access desired for data mining and for many forms of decision support. Instead, the query style is forced to be one of careful planning for a "processing run" in which large chunks of data are temporarily uncompressed, examined as needed, and then compressed back immediately.

The goal of this paper is to develop techniques that will permit the compression of such large datasets in a manner that continues to permit random access to the cells of the matrix. By the term "random access" we mean that the time to reconstruct the value of any single cell is constant with respect to the number of rows N and columns M, with a small proportionality constant. Ideally, it should require 1 or 2 disk accesses (versus 1 disk access that the uncompressed file would require if the whole file could fit on the disk). This is what is required to efficiently support ad hoc queries.

Table 1 provides an example of the kind of matrix that is typical in warehousing applications, where rows are customers, columns are days, and the values are the dollar amounts spent on phone calls each day. Alternatively, rows could correspond to patients, with hourly recordings of their temperature for the past 48 hours, or companies, with stock closing prices over the past 365 days. Such a setting also appears in other contexts. In information retrieval systems rows could be text documents, columns could be vocabulary terms, with the (i, j) entry showing the importance of the j-th term for the i-th document.

day customer	We 7/10/96	Th 7/11/96	Fr 7/12/96	Sa 7/13/96	Su 7/14/96
ABC Inc.	1	1	1	0	0
DEF Ltd.	2	2	2	0	0
GHI Inc.	1	1	1	0	0
KLM Co.	5	5	5	0	0
Smith	0	0	0	2	2
Johnson	0	0	0	3	3
Thompson	0	0	0	1	1

Table 1: Example of a (customer-day) matrix

To make our discussion more concrete, we will refer to rows as "customers" and to columns as "days". The mathematical machinery is applicable to many different applications, such as those mentioned in the preceding paragraph, including ones where there is no notion of a customer or a day, as long as the problem involves a set of vectors or, equivalently, an $N \times M$ matrix $\mathbf{X}$.

Decision support and data mining on large datasets often involves, at the lowest level, obtaining answers to queries, both exploratory queries as well as queries to verify hypotheses. These queries may require access to data records, either individually or in the aggregate: for one, some, or all customers; for one, some, or all days. Two typical queries are:

- Queries on specific cells of the data matrix: '*what was the amount of sales to GHI Inc. on July*

11, 1996?'

- Aggregate queries on selected rows and columns: '*find the total sales to business customers (ABC, DEF, GHI, and KLM) for the week ending July 12, 1996.*'

We study these two main classes of queries in this paper.

There are three underlying assumptions/motivations behind the present work:

- The data matrix is huge, of the order of several GigaBytes. For example, in large corporations like AT&T, there are millions of customers (= rows);
- The number of rows N is much larger than the number of columns M:

  $$N \gg M \tag{1}$$

  As mentioned, N is on the order of millions; we expect that the number of columns M is of the order of hundreds. For example, M=365 if we maintain daily data for a year's duration and 10*12 if we maintain monthly data for the last decade;
- There are no updates on the data matrix, or they are so rare that they can be batched and performed off-line.

In this paper we explore the application of a variety of lossy compression techniques that permit quick reconstruction of arbitrary parts of the dataset. We find that Singular Value Decomposition of the given data matrix, followed by retention of only the few most important principal components, works rather well, resulting in a compressed version that can be used to reconstruct an arbitrary value with only one disk look-up, and with small average error in the reconstructed value. We develop an enhanced algorithm, which we call SVDD, that exhibits not only a smaller average reconstruction error than SVD (and the other compression techniques we tried), but also a very good bound on the error of the data value reconstructed worst. We present computation of SVDD with only three passes over the matrix.

No previous work, to our knowledge, has addressed the problem we study in this paper, even though work on data compression abounds. Some interesting work has been done on compression with fast searching in a large database of bit vectors [12, 5]. Our work is different because our focus is on a dataset of real-valued numbers rather than bit vectors.

Well-designed index structures are necessary to support ad hoc queries. There has been much work on index structures, including some excellent recent work specifically aimed at decision support [8, 10]. However, the design of indices is not the focus of this paper. Our concern is actually getting the data records once they have been identified, which we expect would typically be by means of an index, but could, for the purposes of this paper, be by any other means just as well.

The paper is organized as follows: Section 2 gives the survey. Section 3 describes the mathematical background for the singular value decomposition (SVD). Section 4 gives the algorithms and the proposed enhancements. Section 5 gives experimental results on real datasets. Section 6 makes some observations. Section 7 lists the conclusions and directions for future research.

2 Survey - Alternative Methods

The problem we address in this paper is the compression of a set of time sequences (or vectors), in a potentially lossy manner, while maintaining “random access”, that is, fast reconstruction of any desired cell of the matrix. Several popular data representation techniques from different areas come to mind, including (lossless) string compression, Fourier analysis, clustering, and singular value decomposition (SVD). We examine the first three in the next three subsections, and present SVD in detail in the next section.

2.1 String Compression

Algorithms for lossless string compression are widely available (*e.g.*, `gzip`, based on the well-known Lempel-Ziv algorithm [29], Huffman coding, arithmetic coding, *etc.*; see [23]). While these techniques can achieve fairly good compression, the difficulty with them has to do with reconstruction of the compressed data. Given a query that asks about some customers or some days, we have to uncompress the entire database, for all customers and all days, to be able to answer the query. When there is a continuous stream of queries, as one would expect in data analysis, it effectively becomes the case that the data is retained uncompressed much (or all) of the time.

One attempt to work around this problem is to segment the data and then compress each segment independently. If the segments are large enough, good compression may be achieved while making it sufficient to uncompress only the relevant segments. This idea works only if most queries follow a particular form that matches the segmentation. For truly ad hoc querying, as is often the case in data analysis, such segmentation is not effective. A large fraction of the queries cut across many segments, so that large fractions of the database have to be reconstructed.

For the above reasons, we do not examine lossless compression methods in more detail here.

2.2 Clustering

A different approach is to exploit the observation that the behavior of many customers is likely to be similar. If similar customers can be clustered together, a single cluster representative could serve as a good approximation of the others. Other customers need only have a reference to specify the correct cluster representative. Reconstruction in this case is particularly simple: To find the value of cell $x_{i,j}$, find the cluster-representative for the i-th customer, and return its j-th entry. This application of clustering is known in the signal processing literature as *vector quantization* [16].

Clustering has attracted tremendous interest, from diverse fields and for diverse applications: in information retrieval for grouping together documents represented as vectors [20]; in pattern matching, for grouping together samples of the training set [3]; in the social and natural sciences for statistical analysis [9]. Excellent surveys on clustering include [18, 13, 26].

Although useful in numerous applications, in our setting clustering might not scale-up. The so-called “sound” clustering algorithms, which presumably give the highest quality clusters [26], are typically $O(N^2)$ or $O(N \log N)$. Faster, approximate algorithms include the popular “k-means” algorithm [17], which requires a constant, but large number of passes over the dataset, thus becoming impractical for the huge datasets we have in mind. Recent fast clustering algorithms for huge databases include CLARANS [14], BIRCH [28], and CLUDIS [6]. However, these have only been tried for $M=2$ dimensions. They will probably suffer in high dimensionalities (*e.g.*, $M \approx 100$), if they are based on R*-trees [6] or any other related spatial access method [28].

In our experiments we used an off-the-shelf clustering method from the 'S' statistical package [2]. The method is quadratic on the number of records N, and it builds a cluster-hierarchy, which we truncate at the appropriate levels, to obtain the desirable number of clusters. We set the distance function to be the Euclidean distance, and the "element-to-cluster" distance function to be the maximum distance between the element and the members of the cluster. This results in many tight clusters, which should lead to small reconstruction error. The package had no problems with high dimensions, at the expense of its inability to scale-up for large N.

2.3 Spectral Methods

The lossy spectral representation of real time sequences has been studied extensively in the signal processing literature. Fourier analysis is perhaps the best known of the standard techniques, although there is a plethora of other techniques, such as wavelets [19], linear predictive coding [16], and so forth.

Consider Fourier analysis, where a given time signal is "transformed" to obtain a set of Fourier coefficients. In many practical signals, it is the case that most of the "energy" (or "information") is concentrated in the first few Fourier coefficients [21]. One can then throw away the remaining coefficients. This effect has also been observed in the data mining context[1].

The DFT and other associated methods (*e.g.*, DCT, DWT) are all linear transformations, which effectively consider an M-long time sequence as a point in M-d space, and rotate the axes. This is exactly what the proposed SVD does, but in an *optimal* (in the sense of L_2-norm approximation) way for the given dataset (Figure 1 gives an illustration). Thus, we expect that all these methods will be inferior to the proposed SVD. This is the main reason that we don't put much emphasis on the spectral methods. Additional reasons are the following:

- Spectral methods are tuned for time sequences, ideally with a few low-frequency harmonics. Thus, they won't perform well if the input signals have several spikes or abrupt jumps. Therefore, one should expect SVD to handle discontinuities better than spectral methods.
- The SVD can be applied not only to time sequences, but to any arbitrary, even heterogeneous, M-dimensional vectors. For example, a patient record could be a "vector" comprising elements age, weight, height, cholesterol level, *etc.*. In such a setting, the spectral methods do not apply.

Alternatively, we could treat our two-dimensional matrix as a "photograph image", the values of the cells being the gray-scale values, and apply ideas from two-dimensional signal processing, such as a 2-D Fourier Transform. This is a bad idea because one is now transforming the entire dataset globally, and this is clearly worse than doing it a row at a time: The reason is that adjacent customers need not be related, making the columns look like white-noise signals, which are the worst case for compression. Also, reconstruction of any chosen data cell requires more work.

In conclusion, spectral methods on a row-basis are a good idea; however, their reconstruction performance will never exceed the one for SVD, which constitutes the optimal linear transformation for a

Symbol	Definition
N	number of records/time sequences
M	duration (length) of each sequence
k	cutoff (number of principal components retained during compression)
$\mathbf{X}$	the $N \times M$ data matrix
$\hat{\mathbf{X}}$	the $N \times M$ reconstruction of the data matrix
r	rank of the data matrix
$\mathbf{\Lambda}$	diagonal matrix with eigenvalues
$\|\|\cdot\|\|_2$	Euclidean (= L_2) norm
$\times$	matrix multiplication
$\mathbf{X}^t$	the transpose of $\mathbf{X}$
$x_{i,j}$	value at row i and column j of the matrix $\mathbf{X}$
$\hat{x}_{i,j}$	reconstructed (approximate) value at row i and column j
$x_{i,*}$	the i-th row of the matrix $\mathbf{X}$
$x_{*,j} \equiv \mathbf{x_j}$	the j-th column of the matrix $\mathbf{X}$
$\bar{x}$	the mean cell value of $\mathbf{X}$
γ_i	number (=count) of outlier cells for which deltas are stored in SVDD given that i principal components have been retained
$RMSPE$	normalized root mean squared error
$s\%$	disk space after compression, per cent of original

Table 2: Symbols, definitions and notation from matrix algebra.

given dataset. In our experiments, we use DCT as representative of the spectral methods because it is very close to optimal when the data is correlated [7, p. 109], as is the case in our datasets.

3 Introduction to SVD – The Proposed Method

The proposed method is based on the so-called *Singular Value Decomposition (SVD)* of the data matrix. SVD is a popular and powerful operation, and it has been used in numerous applications, such as statistical analysis (as the driving engine behind the *Principal Component Analysis* [11]), text retrieval under the name of *Latent Semantic Indexing* [4], pattern recognition and dimensionality reduction as the Karhunen-Loeve (KL) transform [3], and face recognition [25]. SVD is particularly useful in settings that involve least-squares optimization such as in linear regression, dimensionality reduction, and matrix approximation. See [24] or [15] for more details. The latter citation also gives 'C' code.

3.1 Preliminaries

We shall use the following notational conventions from linear algebra:

- Bold capital letters denote matrices, *e.g.*, $\mathbf{U}$, $\mathbf{X}$.
- Bold lower-case letters denote *column* vectors, *e.g.*, $\mathbf{u}$, $\mathbf{v}$.

- The "×" symbol indicates explicitly the multiplication of two matrices, two vectors, or a matrix and a vector.

Table 2 gives a list of symbols and their definitions

The SVD is based on the concepts of eigenvalues and eigenvectors:

Definition 3.1 *For a square $n \times n$ matrix $\mathbf{S}$, the unit vector $\mathbf{u}$ and the scalar λ that satisfy*

$$\mathbf{S} \times \mathbf{u} = \lambda \times \mathbf{u} \tag{2}$$

are called an eigenvector and its corresponding eigenvalue of the matrix $\mathbf{S}$.

3.2 Intuition behind SVD

Before we give the definition of SVD, it is best that we try to give the intuition behind it. Consider a set of points as before, represented as an $N \times M$ matrix $\mathbf{X}$. In our running example, such a matrix would represent for N customers and M days, the dollar amount spent by each customer on each day. It would be desirable to group similar customers together, as well as similar days together. This is exactly what SVD does, automatically! Each group corresponds to a "pattern" or a "principal component", *i.e.*, an important grouping of days that is a "good feature" to use, because it has a high discriminatory power and is orthogonal to the other such groups.

Figure 1 illustrates the rotation of axis that SVD implies: suppose that we have M=2 dimensions; then our customers are 2-d points, as in Figure 1. The corresponding 2 directions (x' and y') that SVD suggests are shown. The meaning is that, if we are allowed only k=1, the best direction to project on is the direction of x'; the next best is y', *etc.*

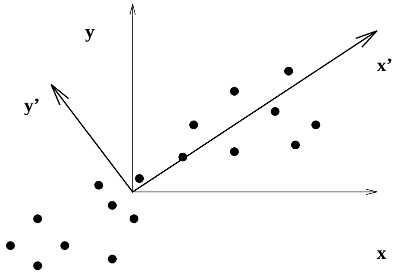


Figure 1: Illustration of the rotation of axis that SVD implies: the "best" axis to project is x'.

3.3 Definition of SVD

The formal definition for SVD follows:

Theorem 3.1 (SVD) *Given an $N \times M$ real matrix $\mathbf{X}$ we can express it as*

$$\mathbf{X} = \mathbf{U} \times \mathbf{\Lambda} \times \mathbf{V}^t \tag{3}$$

where $\mathbf{U}$ is a column-orthonormal $N \times r$ matrix, r is the rank of the matrix $\mathbf{X}$, $\mathbf{\Lambda}$ is a diagonal $r \times r$ matrix and $\mathbf{V}$ is a column-orthonormal $M \times r$ matrix.

Proof: See [15, p. 59]. □

Recall that a matrix $\mathbf{U}$ is called *column-orthonormal* if its columns $\mathbf{u}_i$ are mutually orthogonal unit vectors. Equivalently: $\mathbf{U}^t \times \mathbf{U} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Also, recall that the rank of a matrix is the highest number of linearly independent rows (or columns).

Eq. 3 equivalently states that a matrix $\mathbf{X}$ can be brought in the following form, the so-called *spectral decomposition* [11, p. 11]:

$$\mathbf{X} = \lambda_1 \mathbf{u}_1 \times \mathbf{v}_1^t + \lambda_2 \mathbf{u}_2 \times \mathbf{v}_2^t + \ldots + \lambda_r \mathbf{u}_r \times \mathbf{v}_r^t \tag{4}$$

where $\mathbf{u}_i$, and $\mathbf{v}_i$ are column vectors of the $\mathbf{U}$ and $\mathbf{V}$ matrices respectively, and λ_i the diagonal elements of the matrix $\mathbf{\Lambda}$. Without loss of generality, we can assume that the eigenvalues λ_i are sorted in decreasing order. Returning to Figure 1, $\mathbf{v}_1$ is exactly the unit vector of the best x' axis; $\mathbf{v}_2$ is the unit vector of the second best axis, y', and so on.

Geometrically, $\mathbf{\Lambda}$ gives the strengths of the dimensions (as eigenvalues), $\mathbf{V}$ gives the respective directions, and $\mathbf{U} \times \mathbf{\Lambda}$ gives the locations along these dimensions where the points occur.

In addition to axis rotation, another intuitive way of thinking about SVD is that it tries to identify "rectangular blobs" of related values in the $\mathbf{X}$ matrix. This is best illustrated through an example.

Example: For example, for the above "toy" matrix of Table 1, we have two "blobs" of values, while the rest of the entries are zero. This is confirmed by the SVD, which identifies them both:

$$\mathbf{X} = \begin{bmatrix} 0.18 & 0 \\ 0.36 & 0 \\ 0.18 & 0 \\ 0.90 & 0 \\ 0 & 0.53 \\ 0 & 0.80 \\ 0 & 0.27 \end{bmatrix} \times \begin{bmatrix} 9.64 & 0 \\ 0 & 5.29 \end{bmatrix} \times \begin{bmatrix} 0.58 & 0.58 & 0.58 & 0 & 0 \\ 0 & 0 & 0 & 0.71 & 0.71 \end{bmatrix} \tag{5}$$

or, in "spectral decomposition" form:

$$\mathbf{X} = 9.64 \times \begin{bmatrix} 0.18 \\ 0.36 \\ 0.18 \\ 0.90 \\ 0 \\ 0 \\ 0 \end{bmatrix} \times [0.58,\ 0.58,\ 0.58,\ 0,\ 0] + 5.29 \times \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0.53 \\ 0.80 \\ 0.27 \end{bmatrix} \times [0,\ 0,\ 0,\ 0.71,\ 0.71]$$

Notice that the rank of the $\mathbf{X}$ matrix is r=2: there are effectively 2 types of customers: weekday (business) and weekend (residential) callers, and two patterns (*i.e.*, groups-of-days): the "weekday pattern" (that is, the group {'We', 'Th', 'Fr'}), and the "weekend pattern" (that is, the group {'Sa', 'Su'}). The intuitive meaning of the $\mathbf{U}$ and $\mathbf{V}$ matrices is as follows:

Observation 3.1 $\mathbf{U}$ can be thought of as the *customer-to-pattern* similarity matrix,

Observation 3.2 Symmetrically, $\mathbf{V}$ is the *day-to-pattern* similarity matrix.

For example, $v_{1,2} = 0$ means that the first day ('We') has zero similarity with the 2nd pattern (the "weekend pattern").

Observation 3.3 The column vectors $\mathbf{v}_j$ $(j = 1, 2, \ldots)$ of the $\mathbf{V}$ are unit vectors that correspond to the directions for optimal projection of the given set of points

For example, in Figure 1, $\mathbf{v}_1$ and $\mathbf{v}_2$ are the unit vectors on the directions x' and y', respectively.

Observation 3.4 The i-th row vector of $\mathbf{U} \times \mathbf{\Lambda}$ gives the coordinates of the i-th data vector ("customer"), when it is projected in the new space dictated by SVD.

Lemma 3.2 *The matrix* $\mathbf{C} = \mathbf{X}^t \times \mathbf{X}$ *is a symmetric matrix, whose eigenvalues are the squares of the* λ_i *elements of the* $\mathbf{\Lambda}$ *matrix of the SVD of* $\mathbf{X}$*. Moreover, the columns of the* $\mathbf{V}$ *matrix are the eigenvectors of the* $\mathbf{C}$ *matrix.*

$$\mathbf{C} = \mathbf{V} \times \mathbf{\Lambda}^2 \times \mathbf{V}^t \tag{6}$$

Proof: See [7]. □

The intuitive meaning of the $M \times M$ matrix $\mathbf{C} = \mathbf{X}^t \times \mathbf{X}$ is that it gives the column-to-column similarities. In our example, we have the day-to-day similarities:

$$\mathbf{C} = \mathbf{X}^t \times \mathbf{X} \begin{bmatrix} 31 & 31 & 31 & 0 & 0 \\ 31 & 31 & 31 & 0 & 0 \\ 31 & 31 & 31 & 0 & 0 \\ 0 & 0 & 0 & 14 & 14 \\ 0 & 0 & 0 & 14 & 14 \end{bmatrix}$$

A symmetric lemma can be defined with respect to a "row-to-row similarity" matrix $\mathbf{R} = \mathbf{X} \times \mathbf{X}^t$. We do not present this lemma since it is not required below, whereas Lemma 3.2 is the basis of the two-pass algorithm for the computation of the SVD, which we present is subsection 4.1.

3.4 Outline of Proposed Method

In conclusion, the proposed method is to use the SVD of the data matrix $\mathbf{X}$ (see Eq. 4),

$$\mathbf{X} = \sum_{i=1}^{r} \lambda_i \mathbf{u}_i \times \mathbf{v}_i^t \tag{7}$$

and truncate to the first k few terms $(k \leq r \leq M)$:

$$\hat{\mathbf{X}} = \sum_{i=1}^{k} \lambda_i \mathbf{u}_i \times \mathbf{v}_i^t \tag{8}$$

The idea is to keep as many eigenvectors as the space restrictions permit. The retained terms are known as the k *principal components*. We refer to this method as "SVD" for the rest of this work, or as "plain SVD" (in light of the upcoming enhancements).

The original matrix $\mathbf{X}$ comprise $N * M$ data elements; the SVD representation, after truncating to k principal components, will need $N * k$ data elements for the $\mathbf{U}$ matrix, k data elements for the eigenvalues, and $k * M$ data elements for the $\mathbf{V}$ matrix. Thus the ratio s of space-after over space-before is

$$s = \frac{N * k + k + k * M}{N * M} \approx \frac{k}{M} \tag{9}$$

where the approximation holds, since $N \gg M \geq k$.

4 Algorithms and Enhancements: SVDD

In this section, we first present the algorithms for the "plain SVD" method outlined above; afterwards, we propose an enhancement, the "SVDD" method, which gives much better performance.

4.1 Algorithms for Plain SVD

Here we describe the efficient implementation of SVD for large matrices. Specifically, we present a fast, 2-pass algorithm to compute the $\mathbf{U}$, $\mathbf{\Lambda}$ and $\mathbf{V}$ matrices. We also discuss how to reconstruct a desired cell $(i,\ j)$ from the compressed structure. For the discussion below, recall that k is the number of eigenvalues (and eigenvectors) retained. Typically, $k \ll M$, resulting in a matrix $\mathbf{U}$ much smaller than the original matrix $\mathbf{X}$.

2-pass computation of SVD: We show that it takes only two passes over the large data matrix to compute the SVD, assuming that there is enough memory to hold the $M \times M$ column-to-column similarity matrix $\mathbf{C}$. The idea is to exploit Lemma 3.2. The lemma tells us that we can work with the smallest dimension (M, in our case), compute the $M \times M$ column-to-column similarity matrix $\mathbf{C}$ (which can be done in a single pass), and then compute its eigenvectors (*i.e.*, the $\mathbf{V}$ matrix) and its eigenvalues (*i.e.*, the square of the $\mathbf{\Lambda}$ matrix), in main-memory, since the $\mathbf{C}$ matrix is small. Then only one more pass is required to determine $\mathbf{U}$, as explained next.

Computation of $\mathbf{C}$: During the first pass, we construct $\mathbf{C}$. This is done by keeping track of the partial sum of each element of $\mathbf{C}$. One row ($= M$ elements) of $\mathbf{X}$ is read in at a time, after which every combination of two elements in that row is multiplied and added to the appropriate element of $\mathbf{C}$. Pseudocode for the algorithm is given in Figure 2.

Computation of $\mathbf{U}$: Given that $\mathbf{C}$ is in main memory, we find its eigenvalues and eigenvectors, by Lemma 3.2: $\mathbf{C} = \mathbf{V} \times \mathbf{\Lambda}^2 \times \mathbf{V}^t$. We are ultimately interested in finding the SVD of the data matrix $\mathbf{X} = \mathbf{U} \times \mathbf{\Lambda} \times \mathbf{V}^t$. Since we already have $\mathbf{\Lambda}$ and $\mathbf{V}^t$, $\mathbf{U}$ can be constructed as follows:

$$\mathbf{U} = \mathbf{X} \times \mathbf{V} \times \mathbf{\Lambda}^{-1} \tag{10}$$

```
/* input: pointer to matrix $X$ on disk */
/* output: column-to-column similarity matrix $C$ */

for i := 1 to $M$ do
    for j := 1 to $M$ do
        C[i][j] ← 0;
for i := 1 to $N$ do
    Read $i$-th row of X from the disk (X[i][1], ..., X[i][M])
    for j := 1 to $M$ do
        for l := 1 to $M$ do
            C[j][l] += X[i][j]*X[i][l];
```

Figure 2: Algorithm for computing the column-to-column similarity matrix **C** in one pass.

or, equivalently:

$$u_{i,j} = \sum_{m=1}^{M} x_{i,m} * v_{m,j} / \lambda_j \quad i = 1, \ldots, N; \quad j = 1, \ldots, k \tag{11}$$

Pseudocode for this is given in Figure 3. Notice that the computation of the i-th row $u_{i,*}$ of the matrix **U** needs only the i-th row $x_{i,*}$ of the data matrix **X** (as well as the matrix **V** and the eigenvalues, which are assumed to be in main memory). This is the reason that we need *only one more pass* over the rows of the **X**, in our goal to compute and print **U**.

```
/* input: pointer to $X$ on disk, eigenvectors (V matrix), and eigenvalues $\lambda_j$ */
/* output: row-to-pattern similarity matrix U */

for i := 1 to $N$ do
    Read X[i][*] from disk;     /* row vector of X */
    for j := 1 to $k$ do
        U[i][j] ← 0;
        for l := 1 to $M$ do
            U[i][j] += X[i][l]*V[l][j];
        U[i][j] ← U[i][j] / $\lambda_j$;
```

Figure 3: Algorithm for computing the "row-to-pattern similarity matrix" **U**.

Reconstruction: Given the truncated **U**, **Λ** and **V** matrices, we can derive the reconstructed value $\hat{x}_{i,j}$ of *any* desired cell (i, j) of the original matrix using Eq. 8, or, identically:

$$\hat{x}_{i,j} = \sum_{m=1}^{k} \lambda_m * u_{i,m} * v_{j,m} \quad i = 1, \ldots, N; \quad j = 1, \ldots, M \tag{12}$$

This requires $O(k)$ compute time, independent of N and M. Assuming that **V** and **Λ** are already pinned in memory, that the matrix **U** is stored row-wise on disk, and that an entire row fits in one disk block, only a single disk access is required to perform this reconstruction.

4.2 Proposed Enhancement: SVD with Deltas

There is always the possibility that some data may be approximated poorly. By storing this information separately, we can establish a bound on the error of any individual data element, and also get a reduction in the overall error.

We choose the cells for which the SVD reconstruction shows the highest error, and maintain a set of triplets of the form (*row, column, delta*), where *delta* is the difference between the actual value and the value that SVD reconstructs. The motivation is that a given customer may follow the patterns that SVD expects, with a few deviations on some particular days. Thus, it is more reasonable to store the deltas for those specific days, as opposed to treating the whole customer as an outlier. With this, one can "clean up" any gross errors that the SVD algorithm may have been unable to handle. We call the resulting method, *"SVDD"*, for "SVD with Deltas".

The practical question that arises is how much storage to allocate for keeping outlier information. In other words, we have to tradeoff the number k of principal components retained against the number of data cells that can be considered outliers. Formally, we must solve the following problem:

Given: a desired compression ratio (say, compressed size $s\%$ of the original)

Find: the optimal number of principal components k_{opt} to keep,

Such That: the total reconstruction error is minimized when we are allowed to store cell-level deltas.

Let k_{max} be the largest value of k that does not violate the space requirement, and γ_k be the count of outlier cells that we can afford to store, after we have chosen to maintain k eigenvalues. A straightforward, inefficient way to proceed is given in Figure 4.

```
/* input:  pointer to $X$ on disk, $k_{max}$
/* output:  $k_{opt}$ */

    for $k$ := 1 to $k_{max}$ do
        determine the number of outliers $\gamma_k$ we can afford to store;
        compute the SVD of the array with given $k$ (two passes);
        find the errors for every cell;
        pick the $\gamma_k$ largest ones (one more pass) and
        compute the error measure $\epsilon_k$
    $k_{opt}$ ← value of $k$ with the smallest error measure $\epsilon_k$
```

Figure 4: Straightforward, inefficient algorithm for SVDD

We can factor out several passes and do the whole operation in three passes rather than $3 * k_{max}$. The idea is to create priority queues for the deltas (one queue for each candidate value of k), and to compute all the necessary deltas for all the queues in a single pass over the data matrix. Figure 5 presents pseudocode.

```
input: pointer to $X$ on disk, $k_{max}$
output: matrices $\mathbf{\Lambda}$, $\mathbf{V}$, and $\mathbf{U}$


pass 1:
    compute $\mathbf{\Lambda}$ and $\mathbf{V}$, keeping $k_{max}$ eigenvalues
    estimate the number of outliers $\gamma_k$ that we can afford to store to stay within $s$%, for
    $k$=1, 2, ..., $k_{max}$
    initialize $k_{max}$ priority queues to store the $\gamma_k$ largest cell-outliers for each
    candidate value of $k$
pass 2: for each row of the data matrix,
    compute the error of each cell according to $k$=1,2, ..., $k_{max}$ eigenvalues;
    insert the appropriate cells into the appropriate priority-queue;
    accumulate the reconstruction error $\epsilon_k$ for each $k$ value, so far.
    $k_{opt} \leftarrow$ the $k$ value that gives the smallest error $\epsilon_k$;
    using $k_{opt}$ as the chosen cut-off value $k$, truncate $\mathbf{\Lambda}$ and $\mathbf{V}$;
pass 3:
    pass through each row of the data matrix, to compute and print the corresponding row
    of $\mathbf{U}$, using Eq. 11.
```

Figure 5: 3-pass algorithm for SVDD.

Note that the definition of the reconstruction error is orthogonal to the SVDD algorithm. We continue to use the sum of squared errors (see Eq. 13) as our error metric, as in the rest of this paper.

Data structures for SVDD: Clearly, we need to store $\mathbf{U}$, the k_{opt} eigenvalues, and $\mathbf{V}$, as in the plain SVD. In addition, we have to store the $\gamma_{k_{opt}}$ triplets of the form (*row, column, delta*) for the outlier cells. This should be done in a hash table, where the key is the combination of (row*M+$column$), that is, the order of the cell in the row-major scanning. Optionally, we could use a main-memory Bloom filter [22], which would predict the majority of non-outliers, and thus save several probes into the hash table.

Reconstruction: Reconstructing the value of a single cell, say (i, j) now requires:

- one disk access to fetch the i-th row of $\mathbf{U}$ (as in plain SVD), and then k_{opt} main memory operations with $\mathbf{\Lambda}$ and $\mathbf{V}$, to reconstruct the value of the cell that plain SVD would have reconstructed, using Eq. 8 or 12;
- one probe of hash table to find whether this cell was an outlier, in which case we add the corresponding delta value, and enjoy error-free reconstruction.

5 Experiments

We consider two types of queries in our experiments: queries that seek a specific data value, for a specific customer and a specific day; and queries that seek an aggregate over a set of customers and a set of days. Clearly, these are not the only query types supported by the techniques just described. However, we use these two classes of queries as representative.

We ran our experiments on a variety of real and synthetic datasets. We present here results from two real datasets. The general trends were similar in the other datasets. Following is a description of them.

'phone100K' The first dataset is business data (specifically, AT&T customer calling data). For a selected set of customers, it contains the daily call volume over some period. Given a large enough number of customers and a long enough period of interest, the size of this dataset makes it extremely unmanageable for data analysis. We have N=100,000 customers and M=366 days (a leap year) for each. The size of **'phone100K'** is 0.2 GigaBytes. We also used subsets of this dataset, called **'phone1000'** (1000 rows), **'phone2000'** (2000 rows), *etc.*

'stocks' The second dataset is a list of daily stock closing prices for a number of stocks, again over a specific period. There are N=381 stocks, with M=128 days each. The size of **'stocks'** is 341 KBytes.

Methods: We used plain SVD, the proposed SVDD (SVD with deltas), the hierarchical clustering method (described in Sec. 2.2), and the Discrete Cosine Transform (DCT) from the spectral methods (Sec. 2.3).

Error measure: There are many different measures that one could use for reconstruction error, based on different application needs. The root-mean-squared-error (absolute or relative) is the typical error measure for forecasting applications in time series [27]. We use this metric, one again, normalized with respect to the standard deviation of the data values being recorded. We call this the root mean square percent error (RMSPE).[1]. .

For any method, let $\hat{x}_{i,j}$ be the reconstructed value of the i, j cell, when the original value was $x_{i,j}$ and let $\bar{x}$ be the mean cell value of $\mathbf{X}$.

Definition 5.1 *The RMSPE is defined as the normalized root mean squared error:*

$$\frac{\sqrt{\sum_{i=1}^{N}\sum_{j=1}^{M}(\hat{x}_{ij} - x_{ij})^2}}{\sqrt{\sum_{i=1}^{N}\sum_{j=1}^{M}(x_{ij} - \bar{x})^2}} \tag{13}$$

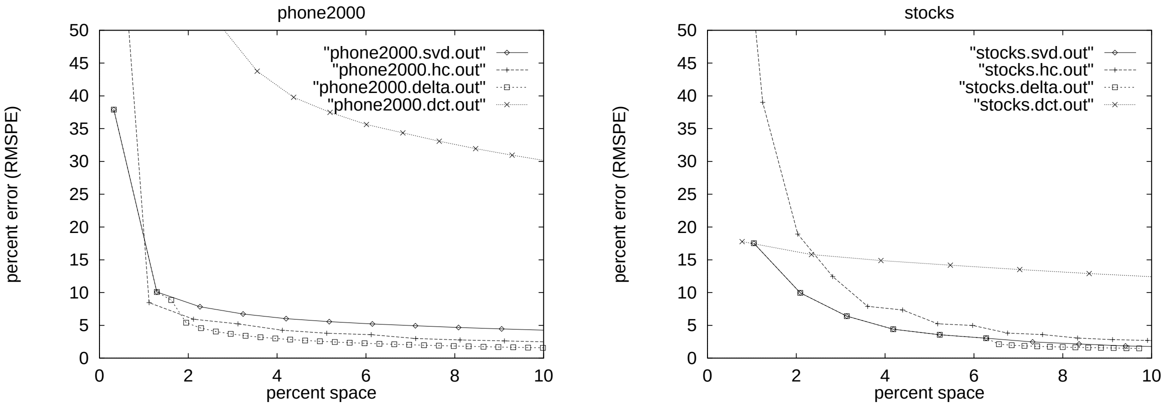
---

[1] The signal processing community uses the signal strength (mean squared amplitude) as a standard normalization factor (for example, in computing "signal-to-noise ratio"). By analogy, our first instinct was to divide by the root-mean-squared x_{ij}. However, our sequence, unlike audio or electro-magnetic signals, do not have zero mean. So we have chosen to subtract out the mean, thereby computing the standard deviation rather than signal strength in the denominator. Note that the choice of normalizing constant does not affect the trends discovered in the experiments, but does impact the magnitude of the normalized error values reported. If we had used signal strength rather than deviation as our normalization, the results we report would appear even better

We ran three sets of experiments: The first was to determine the accuracy vs. space tradeoff for the competing methods. The second was to see how the error changes for aggregate queries involving multiple cells. The last was to see how our method scales up with dataset size. These are the topics of the upcoming subsections, correspondingly.

5.1 Accuracy vs. Space Trade-off

Here we compare the reconstruction error vs. required storage space of four compression methods: hierarchical clustering, DCT, SVD, and SVDD with b bytes of storage space for each number stored. For the clustering method, we store the cluster centroids and an array containing the cluster number to which each point belongs. If there are k clusters to be stored, then $(b \times k \times M) + (N \times b)$ bytes are required. For DCT, we store the low-frequency coefficients; if k coefficients are kept for each row, then $N \times k \times b$ space is required. The space requirements for SVD are given by Eq. 9; the space requirements for SVDD involve the same formula (but for fewer PCs) and then $O(b)$ bytes for each delta stored. To make all results comparable, we present storage space required not in absolute terms, but rather as a percentage (s%) of the storage required uncompressed.



(a) **‘phone2000’** (b) **‘stocks’**

Figure 6: Reconstruction error (RMSPE) vs. disk storage space (s%) for clustering (“$+$”), DCT (“$\times$”), SVD (“$\diamond$”), and SVDD (“$\square$”). The SVD and SVDD curves overlap for low values of s (= the percent space consumed).

Figure 6 plots the reconstruction error (RMSPE) that the competing methods require, as a function of the ratio of disk space to store the compressed format compared to storing the entire matrix. The left graph is for the **‘phone2000’** dataset, while the right one for the **‘stocks’** dataset. The labels “svd’, “delta”, “dct” and “hc” correspond to the plain SVD, the SVD with deltas, DCT, and the hierarchical clustering method as described earlier in subsection 2.2. The observations are as follows:

- In both cases, the proposed SVDD method did the best.
- DCT did not do well. In the **‘phone2000’** case, it consistently had the highest reconstruction error. For stocks prices, which are modeled well as random walks [21], it is believed to be the best among the spectral methods, exactly because successive stock prices are highly correlated.

storage space	SVD (abs error)	SVDD (abs error)	SVD (normalized)	SVDD (normalized)
5%	1794.917	53.745	465.4%	13.93%
10%	1268.717	26.464	328.9%	6.86%
15%	635.456	16.8	164.7%	4.35%
20%	472.784	11.82	122.6%	3.06%
25%	404.824	10.546	104.9%	2.73%

Table 3: Worst-case error as a function of storage space for the **'phone2000'** dataset, both in absolute and in normalized $\frac{ABS}{Std.Dev.}$ terms.

This explains why DCT performs better for the **'stocks'** dataset as opposed to the **'phone2000'** dataset.

- Plain SVD and clustering were close to each other, alternating in the second and third place. Specifically, SVD was better for the **'stocks'** dataset. It should be noted that the clustering method we used was a high-quality quadratic method. Even so, the plain SVD outperformed it or had a comparable reconstruction error. It is questionable how much more reconstruction error a scalable, linear clustering method will lead to, if it can work for $M \approx 100$ at all.
- The SVDD algorithm did best on both datasets, consistently. For very small storage sizes (under $s = 2\%$ for **'phone2000'** and under $s = 6\%$ for **'stocks'**), the optimum value of k_{opt} was k_{max}: that is, it turned out best to devote all the available storage to keeping as many principal components as possible and no outliers.
- For 10% space requirement (*i.e.*, a 10:1 compression ratio), the error was less than 2% for SVDD on both datasets. Even a 50:1 compression ratio ($s = 2\%$) resulted in an error of under 10%. As a point of reference, the Lempel-Ziv (`gzip`) algorithm had a space requirement of $s \approx 25\%$ for both datasets.

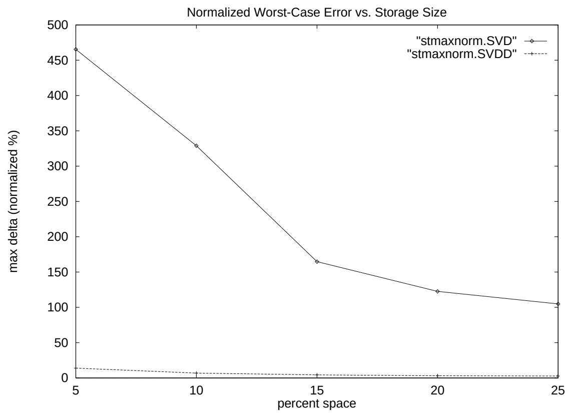


Figure 7: Worst-case error as a function of storage space for the **'phone2000'** dataset.

Thus far, we have used the mean error, RMSPE, as our error metric. However, it is often useful to bound the error on any individual point. We ran some additional experiments on the **'phone2000'** dataset to determine the worst-case error for any one matrix cell. Table 3 shows the results of these experiments for the SVD and SVDD techniques. We plot the maximum error, for a single data point in a time series, as a function of storage space for the **'phone2000'** dataset. Figure 7 presents the same information in a graph. As in the case of RMSPE the error has been normalized with respect to the standard deviation of the dataset.

The results are astounding. Even where the RMSPE is quite reasonable for the plain SVD technique, we find that the worst case error for a single data value can still be very large, potentially causing estimates for selected individual points to be way off. (This is true for the clustering and DCT techniques as well). On the other hand, the SVDD technique bounds the worst error pretty well, so that one can have confidence that the reconstructed value of every single point is within a few percent of the correct value, even with a significant compression ratio.

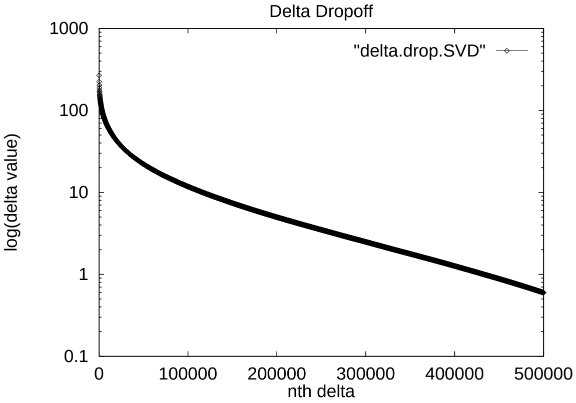


Figure 8: Absolute error vs. deltas ordered by reconstruction error for plain SVD applied to the **'phone2000'** dataset at 10% storage ($k = 31$ principal components).

To understand this phenomenon further, we plotted the distribution of errors for the individual cells. Figure 8 shows the results for the **'phone2000'** dataset, using the SVD technique. The X-axis has the cells rank ordered by the error in their reconstruction, for the first 50,000 cells, and the Y-axis has the absolute error. Observe the steep initial drop in error (the Y-axis is on a logarithmic scale), indicating that only a few points suffer an error anywhere close to the worst-case bound. This is the reason for the good performance of SVDD. By explicitly recording these few very bad cases, it is able to bound the worst case error to under 10% without consuming large amounts of storage. What this means is that, with SVDD, not only do we get good compression with remarkably low average error, but we also know that the worst case cell error is also a close approximation.

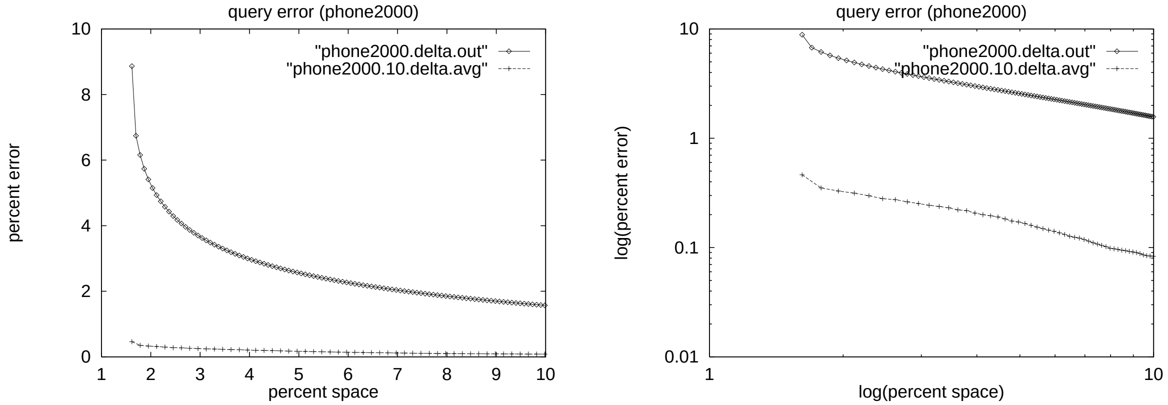
Another observation from Figure 8 is that most matrix cells are reconstructed after compression with an error substantially less than the mean error RMSPE. In applications where a few erroneous data points are tolerable, one may actually care more about the median rather than the mean error, and this median error is one or two orders of magnitude less than the mean error, so all of our techniques actually do much better than one would imagine based on the results for mean error presented above.

For the rest of this work we mainly focus on SVDD, since it performed significantly better than the other techniques.

5.2 Reconstruction Error for Aggregate Queries

The results of the previous experiment are very encouraging: less than 2% error with a 10% space requirement for the SVDD. In fact, the results are even better for aggregate queries because errors tend to cancel out when cell values are aggregated.

In general, an aggregate query specifies some rows and columns of the data matrix and asks for an aggregate function $f()$ of the specified cells (*e.g.*, '*find the sum of sales among our NJ customers, for the 1st of every month in 1995*'). The function $f()$ could be, *e.g.*, *sum*(), *avg*(), *stddev*(), *etc.*



(a) **'phone2000'** linear scales (b) **'phone2000'** logarithmic scales

Figure 9: Query error vs. space overhead for aggregate (avg) queries ("+"): (a) linear and (b) logarithmic scales. The RMSPE (of single-cell queries) is also shown ("◇") for comparison.

For a given aggregate query, we define the *normalized query error* Q_{err} as the relative error between the correct response and our approximate response:

$$Q_{err} = |f(\mathbf{X}) - f(\hat{\mathbf{X}})| / |f(\mathbf{X})| \tag{14}$$

where $f()$ is the aggregate function, over a set of cells that the query specified.

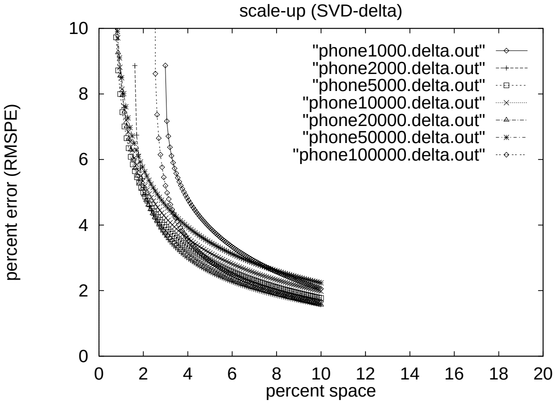
We posed 50 aggregate queries to determine the average of a randomly selected set of rows and columns in the **'phone2000'** dataset. The number of rows and columns selected was tuned so that approximately 10% of the data cells would be included in the selection. Figure 9 presents the results averaged over the 50 queries, showing how error varies as a function of the storage space used. The results are shown for the SVDD method only, since the rest of the methods showed similar behavior. The error was well under 0.5% even with the storage space set to only 2% of the original. In other words, a 50:1 compression ratio can be comfortably obtained. Figure 9 also shows the error for queries on individual cells, that is, the RMSPE that we showed previously.

Estimates of answers to aggregate queries can be obtained through sampling. (Note that sampling is not likely to be able to provide estimates of individual cell values, and is therefore not comparable to the work in the bulk of this paper). In initial experiments we ran, simple uniform sampling performed poorly compared with SVDD for aggregate queries. We did not implement more sophisticated sampling techniques, and an open question is how our techniques compare with sophisticated adaptive sampling for aggregate queries.

5.3 Scale-up

Here we show the reconstruction error for the proposed SVDD method. The current version of the clustering method could not scale up beyond $N = 3000$. We tried using a small sample to do the clustering and then assigning the remaining records to the existing clusters, but this gave very poor results. As mentioned, clustering for large datasets is the topic of recent research (BIRCH [28], CLARANS [14], *etc.*); however, none of these algorithms scales up for high-dimensional points. Thus, we focus our attention on SVDD for the rest of the experiments.

Figure 10 shows the curves for subsets of size N = 1,000, 2,000, 5,000, 10,000, 20,000, 50,000 and the full set of N=100,000 customers from the **'phone100K'** dataset, for the SVDD method.



SVDD.

Figure 10: Reconstruction error (RMSPE) vs. storage space (s%) for SVDD, on the **'phone100K'** dataset

Notice that

- the error is around 2% at the 10% space consumption, for all of the sample sizes;
- the graphs are fairly homogeneous, for a wide span of database sizes (1,000 $\leq N \leq$ 100,000).

As was previously mentioned, the errors for aggregate queries will be even less.

In Table 4 we show how the maximum error changes with the dataset size. We find that for plain SVD the maximum error increases with dataset size. Intuitively, this is because, as the dataset becomes larger, there is a greater likelihood of one bad outlier point that gets reconstructed poorly. However, with the SVDD technique, the maximum error remains approximately constant with dataset size.

6 Discussion

Below we describe how to extend the SVDD algorithm for use with a DataCube, and mention a practical issue. Appendix A discusses the ability of SVD to provide visualization of datasets, essentially for free.

dataset	SVD (normalized)	SVDD (normalized)
‘phone1000’	227.1%	10.6%
‘phone2000’	328.9%	6.8%
‘phone5000’	890.8%	7.9%
‘phone10000’	1306.5%	8.6%
‘phone20000’	1829.6%	7.4%
‘phone50000’	3849.0%	9.4%
‘phone100K’	5335.6%	7.4%

Table 4: Comparison between SVD and SVDD of worst-case normalized errors at 10% storage, for increasing dataset sizes.

6.1 DataCube Compression

Whereas we focus on time sequences in this paper, the techniques described above apply in general to multi-dimensional data. For instance, consider the well-known *productid* × *storeid* × *weekid* array of sales figures. This corresponds to a “DataCube” [8] with three dimensions, while the customer-day dataset we have been discussing corresponds to a DataCube of two dimensions. There are several ways to handle the issue:

- We can group these as *productid* × (*storeid* × *weekid*) or as (*productid* × *storeid*) × *weekid*. Which we prefer is a function of the number of values in each dimension. In general, the more square the matrix, the better the compression, but also the more the work that has to be done to compress. So we pick the largest size for the smaller dimension that still leaves it computable within the available memory resources. The main point to note is that, since the cells in the array are reconstructed individually, how dimensions are collapsed makes no difference to the availability of access.
- This problem has also been examined in the principal component analysis literature, under the name of *3-mode PCA* [11, p. 233]. The idea is again to approximate the given “DataCube” element x_{ijk} with $\sum_{h,l,r} a_{ih} b_{jl} c_{kr} g_{hlr}$ where the parameters a_*, b_*, c_* and g_* are chosen to minimize the error. The 3-mode PCA has been extended, in theory, to N-mode analysis. It is an interesting open question to find out the relative benefits of each alternative.

6.2 Practical Issue

In one of our real datasets (results not reported in this paper), we observed that there were several customers that did not make any purchases at all. An “engineering” solution in this case is to flag all these customers, and build a Bloom filter [22], to help detect them quickly.

7 Conclusions

We have examined the problem of providing fast “random access” capability on a huge collection of time sequences. Our contributions are:

- The formulation of the problem as a lossy compression problem, and the proposal of several tools to solve it, from diverse areas, like signal processing (DFT, *etc.*), pattern recognition and information retrieval (clustering), and matrix algebra (SVD).
- The description of SVD, which is a powerful tool in matrix algebra. We hope that the intuition we tried to provide and the pointers to citations and source code, will make SVD accessible to a wider database audience.
- The enhancement of SVD and the detailed design of SVDD, which has several desirable properties:
  - It achieves excellent compression, with the ability to accurately reconstruct the original data matrix (5% error for a 40:1 compression ratio of the largest dataset).
  - It bounds the worst-case error of any individual data value pretty well. Our experiments showed that the reconstructed value of every single cell is within 10% of the correct value at 10% storage, for datasets of increasing size (unlike plain SVD, whose worst-case error increases with dataset size).
  - Its computation requires only three passes over the dataset, which is very desirable for huge datasets.
  - Like SVD, it naturally leads to dimensionality reduction of the given dataset while still preserving distances well, thus allowing visualization and providing a method of detecting outliers for data analysis.
  - It can handle any arbitrary vectors in addition to time sequences without any additional effort.

We presented experiments on real datasets (stocks, calling patterns of AT&T customers), which highlighted the above claims.

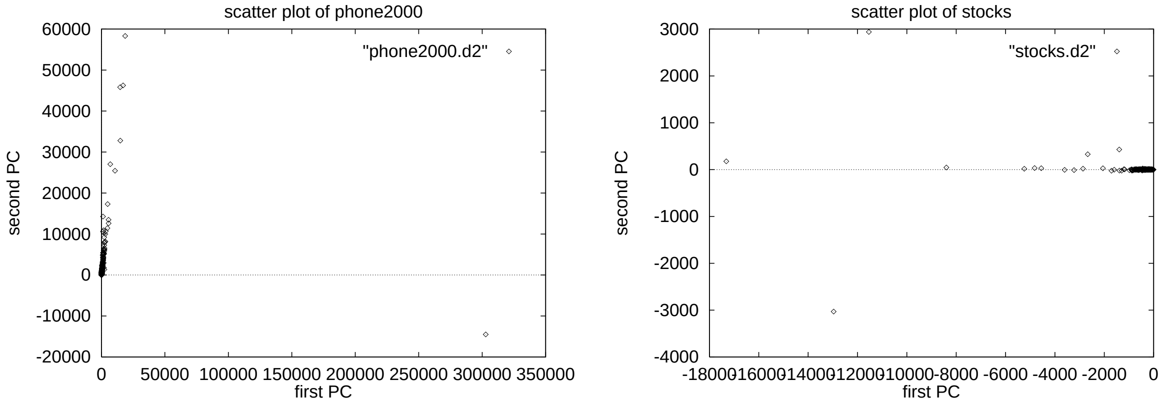
Directions for future research include (a) the design and implementation of robust, scalable clustering algorithms and their comparison against SVDD; (b) the study of the so-called "robust" SVD algorithms (which try to minimize the effect of outliers); and (c) the use of 3-mode and N-mode PCA for DataCube problems.

Acknowledgments:

We would like to thank Ken Church, Rick Greer, and Inderpal Singh Mumick for constructive discussions, and Hans-Peter Kriegel, Raymond Ng, and Xiaowei Xu for providing code on clustering.

A Visualization

Recall that SVD performs a change of axes, and it maintains the first k of them. Thus, we readily have the first 2 or 3 axes, which can be used to map each time sequence into a point in 2- or 3- dimensional space. These points can be plotted to give an idea of the density and structure of the dataset. Figure 11 shows the scatter-plot of the **'phone2000'** and **'stocks'** datasets in the 2-dimensional SVD space, that



(a) **'phone2000'** (b) **'stocks'**

Figure 11: Scatter plot of our two datasets, the **'phone2000'** and the **'stocks'** datasets, in SVD space

is, the first axis corresponds to the first (and strongest) principal component (= linear combination of existing columns), and the second axis corresponds to the second principal component.

There are several observations we can make from the two plots:

- In the **'phone2000'** dataset, most of the points are concentrated close to the origin, with a few exceptions. The exceptions correspond to high-volume customers, while the majority of the customers use the services much less. We could probably make the hypothesis that there is some skewed, Zipf-like distribution of our customers, which could lead to marketing decisions.
- In the **'stocks'** dataset, there is a similar situation but with less skew. Moreover, most of the points are very close to the horizontal axis, implying that they all follow closely the first eigenvector. Presumably, due to the long duration of the sequences, most of the stocks followed the general pattern of the stock market, and thus exhibit similar trends, after scaling.
- The closeness to the horizontal axis explains the excellent compression that we saw in Figure 6(b). It also explains why clustering didn't perform well. Except for a few points close to the origin, the majority of the points are scattered on the horizontal axis. A handful of them are even away from the horizontal axis. Thus, there are no natural clusters to look for, which explains why clustering didn't do well. On the contrary, the **'phone2000'** dataset has almost no points along either of the two axes because the point at approximately (300,000, -15,000) created a large distraction and tilted the axis in an unfavorable way for SVD.
- Presumably, a financial analyst should examine those exceptional stocks whose points are away from the horizontal axis. Also, a marketing analyst could focus on the distraction of the **'phone2000'** dataset.
- In both scatter-plots, there are some outliers. Instead of using additional principal components to achieve better approximations for them, it is much cheaper to store their deltas. This is the reason that SVDD often performs so much better than the plain SVD.

References

[1] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *Fourth Int. Conf. on Foundations of Data Organization and Algorithms (FODO)*, pages 69–84, Evanston, Illinois, October 1993. also available through anonymous ftp, from olympos.cs.umd.edu: ftp/pub/TechReports/fodo.ps.

[2] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language.* Wadsworth & Brooks/Cole Advanced Books & Software, Pacific Grove, CA, 1988.

[3] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[4] Susan T. Dumais. Latent semantic indexing (lsi) and trec-2. In D. K. Harman, editor, *The Second Text Retrieval Conference (TREC-2)*, pages 105–115, Gaithersburg, MD, March 1994. NIST. Special publication 500-215.

[5] Susan Eggers and Arie Shoshani. Efficient access of compressed data. In *Proceedings of the 6th VLDB Conference*, volume 6, pages 205–211, 1980.

[6] Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Knowledge discovery in large spatial databases: Focusing techniques for efficient class identification. *Proc. of 4th International Symposium on Large Spatial Databases*, 1995.

[7] Christos Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Inc., 1996. ISBN 0-7923-9777-0.

[8] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Technical Report No. MSR-TR-95-22, Microsoft, 1995.

[9] John A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, 1975.

[10] Ted Johnson and Dennis Shasha. Hierarchical split cube forests for decision support. Technical report, Draft, September 1996.

[11] I.T. Jolliffe. *Principal Component Analysis.* Springer Verlag, 1986.

[12] J. Li, D. Rotem, and H. Wong. A new compression method with fast searching on large databases. In *Proceedings of the 13th VLDB Conference*, volume 13, pages 311–318, Brighton, England, 1987.

[13] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms. *The Computer Journal*, 26(4):354–359, 1983.

[14] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. *Proc. of VLDB Conf.*, pages 144–155, September 1994.

[15] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992. 2nd Edition.

[16] Lawrence Rabiner and Biing-Hwang Juang. *Fundamentals of Speech Recognition.* Prentice Hall, 1993.

[17] Lawrence Richard Rabiner and Bernard Gold. *Theory and Application of Digital Signal Processing.* Prentice-Hall, Englewood Cliffs, N.J., 1975.

[18] Edie Rasmussen. Clustering algorithms. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 419–442. Prentice Hall, 1992.

[19] Mary Beth Ruskai, Gregory Beylkin, Ronald Coifman, Ingrid Daubechies, Stephane Mallat, Yves Meyer, and Louise Raphael. *Wavelets and Their Applications.* Jones and Bartlett Publishers, Boston, MA, 1992.

[20] G. Salton, E.A. Fox, and H. Wu. Extended boolean information retrieval. *CACM*, 26(11):1022–1036, November 1983.

[21] Manfred Schroeder. *Fractals, Chaos, Power Laws: Minutes From an Infinite Paradise.* W.H. Freeman and Company, New York, 1991.

[22] D.G. Severance and G.M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM TODS*, 1(3):256–267, September 1976.

[23] James A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Inc., 1988.

[24] Gilbert Strang. *Linear Algebra and its Applications.* Academic Press, 1980. 2nd edition.

[25] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[26] C.J. Van-Rijsbergen. *Information Retrieval.* Butterworths, London, England, 1979. 2nd edition.

[27] Andreas S. Weigend and Neil A. Gerschenfeld. *Time Series Prediction: Forecasting the Future and Understanding the Past.* Addison Wesley, 1994.

[28] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD '96*, pages 103–114, Montreal, Canada, June 1996.

[29] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, IT-23(3):337–343, May 1977.

Contents