Compressing Large Boolean Matrices Using Reordering Techniques

David Johnson
AT&T Labs – Research
dsj@research.att.com

Shankar Krishnan
AT&T Labs – Research
krishnas@research.att.com

Jatin Chhugani
Johns Hopkins University
jatinch@cs.jhu.edu

Subodh Kumar
Johns Hopkins University
subodh@cs.jhu.edu

Suresh Venkatasubramanian
AT&T Labs – Research
suresh@research.att.com

Abstract

Large boolean matrices are a basic representational unit in a variety of applications, with some notable examples being interactive visualization systems, mining large graph structures, and association rule mining. Designing space and time efficient scalable storage and query mechanisms for such large matrices is a challenging problem.

We present a lossless compression strategy to store and access such large matrices efficiently on disk. Our approach is based on viewing the columns of the matrix as points in a very high dimensional Hamming space, and then formulating an appropriate optimization problem that reduces to solving an instance of the Traveling Salesman Problem on this space.

Finding good solutions to large TSP's in high dimensional Hamming spaces is itself a challenging and little-explored problem – we cannot readily exploit geometry to avoid the need to examine all N^2 inter-city distances and instances can be too large for standard TSP codes to run in main memory. Our multifaceted approach adapts classical TSP heuristics by means of instance-partitioning and sampling, and may be of independent interest. For instances derived from interactive visualization and telephone call data we obtain significant improvement in access time over standard techniques, and for the visualization application we also make significant improvements in compression.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1 Introduction

Consider the following three problems:

- You are visualizing a large and complex three-dimensional geometric model and you would like to have a real-time walkthrough ($\geq$ 20 frames/s update). In order to do this, you need to determine quickly what parts of the model can be seen from a region of space (cell) bounding your current location.
- You work for a major phone company, and you have access to data that tells you which numbers call which numbers. You would like to manage this data to develop graph models of *communities of interest*.
- You have large volumes of data describing various purchases that people make, and you'd like to infer *association rules* from this very large database.

In all of the above problems, the basic unit of data is a large, disk-resident matrix of ones and zeros. In the first case, rows correspond to transitions between view cells, and columns are the primitives (typically collections of triangles) that become visible in moving from one cell to the next. These matrices are very large, having of the order of hundreds of thousands of rows and columns. Representing them and querying them efficiently is a non-trivial problem. In the second case, rows and columns are individual customers, and each entry of the matrix represents a call made from

one person to another. In the third case, rows are customers and columns are products.

In general, our problem is to store the data so that we can efficiently access the information corresponding to a row:

Problem. *Given two sets R, C and a binary relation $M \subseteq R \times C$, store M efficiently such that for any $r \in R$, the set $M(r) = \{c \mid (r, c) \in M\}$ can be retrieved efficiently.*

If M (viewed as a matrix) is sufficiently dense, then representing M as an adjacency matrix is plausible. However, this does not scale well at all; for $|R|, |C| \geq 10^5$, this is already an impractical solution.

A more reasonable option, given that in applications of interest M tends to be sparse, is to use a sparse graph representation. For each $r \in R$, we maintain a list of elements of $M(r)$. This can be done in two ways; we either explicitly enumerate the elements of C, or maintain pointers into a data structure for C. Note that given the scales involved, both approaches will require using offline storage; in one application each element of C can be 10KB, and each $M(r)$ can be on average of size 1000, yielding over 100 GB of needed storage with the first approach and nearly 1 GB of storage with the second (assuming 10^5 rows).

There are tradeoffs between the two approaches; explicit enumeration is wasteful in space due to the replication of data elements, which means that updates to C can be hard. However, the second approach may require making many seeks into a list, in comparison with the first approach where access to $M(r)$ is relatively efficient.

1.1 Paper Organization

Our proposed solution exploits both the superior access time of the first approach and the efficient space usage of the second. We describe it using a simple example in Section 2 and go into more detail in Sections 3 and 4. We survey related work in Section 5. A detailed experimental study follows in Section 6.

2 Problem Formulation

We start with a brief example to illustrate our approach. Consider the relation M depicted in Table 1. This relation is defined between the sets $R = \{A, B, C, D\}$ and $C = [1..16]$. If we wished to retrieve $M(D)$ from disk, we can either make three distinct seeks into C to extract the entries $\{4, 5, 10, 15, 16\}$, or we can perform one seek and scan the entire list, retaining only the relevant entries (we assume that rows are laid out sequentially on disk).

Suppose however we were able to reorder the IDs of C, so that the relation looked like Table 2. Note now that for each row, all the relevant entries are clustered together; in fact $M(B)$ and $M(C)$ can each be retrieved in a single seek and scan with no wasted disk access.

Definition 2.1. *A* run *in a row of a matrix M is a maximal sequence of non-zero entries.*

Going back to Table 1, row C has 3 runs ($\{2, 3\}, \{6\}$ and $\{9\}$). However, after reordering (see Table 2), it has only 1 run. Since each run requires a single seek, we can now define a cost measure for a given relation.

Definition 2.2. *The cost* $\text{runs}(M)$ *of a matrix M is the sum of the number of runs in each of its rows.*

The *reordering problem* can now be stated as:

Problem 2.1 (Matrix Reordering). *Given a binary matrix M, find a matrix M' obtained by permuting the columns of M that minimizes* $\text{runs}(M')$.

Note that minimizing $runs(M')$ not only speeds up access time – as we shall see later in this paper, it may also significantly decrease the space needed to store the matrix.

One special case of the Matrix Reordering problem can be solved efficiently: the question of whether the optimum value for $runs(M')$ equals the number of rows of M that contain non-zeros. This is equivalent to asking whether the matrix has the following well-studied property.

Definition 2.3 (Consecutive-ones Property). *A matrix M is said to have the* consecutive-ones *property if its columns can be permuted such that in the resulting matrix M', all nonzero elements in each row appear consecutively.*

Booth and Lueker [4] showed in 1976 that for a given matrix M, there is a linear time algorithm that determines whether M has the consecutive-ones property and produces the desired permutation if so. Thus, if the relation has the consecutive-ones property, we can reorder the columns on disk so that the elements of each row can be accessed in a single seek. However, this will in general not be possible and minimizing the number of runs when a matrix does not have the consecutive-ones property is hard:

Theorem 2.1. *Matrix Reordering is NP-hard.*

Proof. We demonstrate a reduction from Hamiltonian Path [14, GT39]. Given an undirected graph $G(V, E)$, construct the boolean matrix M whose rows are edges, columns are vertices, and an entry is 1 if the corresponding vertex and edge are adjacent.

Each row has exactly two 1s in it. Consider an edge $e = (u, v)$. If u and v are adjacent in the column order, e contributes a cost of 1 to the total run cost, else it contributes two. Thus each pair of consecutive vertices that share an edge reduce one unit from the maximum run cost $2|E|$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	*		*		*		*	*			*	*	*		*	
B				*			*			*			*			*
C		*	*			*			*							
D				*	*					*					*	*

Table 1: An example relation M between the sets $\{A, B, C, D\}$ and [1..16].

	12	15	5	7	13	16	10	4	14	8	1	11	3	2	6	9
A	*	*	*	*	*					*	*	*	*			
B				*	*	*	*	*								
C													*	*	*	*
D		*	*			*	*	*								

Table 2: The same relation M after reordering the columns

If G has a Hamiltonian path, then there exists a reordering of the columns of M yielding a matrix M' such that $runs(M') = 2|E| - n + 1$. If not, then any reordering must have cost at least $2|E| - n + 2$. Setting $K = 2|E| - n + 1$, the theorem holds. □

Matrix reordering can be related to the Traveling Salesman Problem in the following manner. Recall that a *Hamming space* is a vector space over binary vectors where the sum of two vectors is their component-wise sum, taken over GF(2) (i.e $0 + 1 = 1 + 0 = 1$, $1 + 1 = 0 + 0 = 0$). This space has a norm $\|\cdot\|$, defined as $\|v\| = \sum_i v_i$, and the corresponding *Hamming metric* $d_H(v_1, v_2) = \|v_1 - v_2\|$. A *tour* in a Hamming (or any distance) space is an order in which points are visited, and the cost of the tour is the sum of distances between adjacent points in the tour. For a given space, let T denote the cost of the shortest tour that visits all points.

Now if we view each column of M as a point in a Hamming space, it is easy to relate $runs(M)$ and the cost of a minimum tour.

Theorem 2.2.

$$0.5T \leq \text{runs}(M) \leq T$$

Proof Sketch. Note that each run contributes at most two units to a tour; one $0 \to 1$ transition at the beginning, and one $1 \to 0$ transition at the end. Runs at the beginning or end of a row contribute only one unit. Thus if we sum up the contributions to the tour for any particular dimension (row), we obtain a quantity that is at least equal to, and is at most twice, the number of runs in that row. □

3 Traveling Salesman Heuristics

The traveling salesman problem is NP-hard even for instances with metric distance functions. The best polynomial-time approximation guarantee known for such instances is 1.5, proved for the $O(N^3)$ Christofides algorithm [10]. Such a running time is infeasible for our applications, but many significantly faster heuristics are known to perform well in practice, typically getting much closer to optimal than promised by the above bound [17]. Four commonly considered such heuristics are

NN: Starting at an arbitrary point, move to its nearest neighbor and repeat. If at any point the nearest neighbor of the current point is in the tour, pick the next nearest neighbor.

2-OPT: Start with NN tour. Pick any two edges (u, v) and (w, x), and delete them, reconnecting with edges (u, x) and (w, v) if the resulting tour has lower cost. Repeat.

3-OPT: Similar to 2-OPT, except that three edges are broken, and the tour is reconstructed in one of two different ways.

Lin-Kernighan: This is a sophisticated algorithm that can perform k-OPT moves for arbitrarily large k, but in a highly structured way that keeps the worst-case running time polynomial.

For random Euclidean instances as studied in [17], NN typically gets within 25% of optimal and the implementations of 2-OPT, 3-OPT, and Lin-Kernighan described therein get within 5%, 3%, and 2% respectively. For the instances we study here, Lin-Kernighan performed too many distance calculations to be run effectively, so it and still-more-sophisticated heuristics were ruled out. On the other hand, results from the other three heuristics were much closer together than for Euclidean instances. 3-OPT was never more than 7% better than NN and, where we could test it, Lin-Kernighan offered little further improvement. Thus restricting ourselves to the first three algorithms should allow us to quantify a realistic range of running-time/solution-quality tradeoffs.

The reason that distance calculations play such a key role in running time here (as opposed to the case of Euclidean instances for example) is that they can be much more expensive. In a high-dimensional Hamming space where each vector can have thousands of

non-zero entries, computing the distance between two points will require thousands of operations, compared to just 6 for the Euclidean distance in two dimensions, and even though the latter may be individually more expensive (involving multiplications and square roots), the former can still add up to much more work. Even high-dimensional Euclidean spaces can be projected to a smaller number of dimensions using standard embedding methods; as we will discuss in more detail later, this is not likely to help for Hamming spaces.

Our implementations of 2- and 3-OPT mitigate this somewhat by first computing a list of 50 nearest neighbors for each city and caching the distances to them. This data structure is also used to guide the search for improving moves, as first suggested by Lin and Kernighan [20], and once constructed can also speed the computation of the NN tour.

Unfortunately, we know of no way to construct the data structure without looking at all $N(N-1)/2$ inter-city distances, which is prohibitively expensive for large N. Another problem with large N is that the data structure or the instance itself may not fit in main memory, which would result in substantial additional costs in terms of disk I/O. We thus must resort to a classic technique for dealing with large TSP's.

3.1 Splitting The Tour

Given a TSP instance, instead of computing a tour on the entire input we can partition the cities into two sets, compute a tour for each part, and concatenate the two partial tours. Since neighbor-list construction is at least quadratic in the input size, this approach will speed up the TSP computation process, presumably at the cost of a decrease in the quality of the combined tour. In general, we can break up the input into k pieces, solve the TSP on each piece, and then glue the pieces together. In addition to speeding up the individual TSP calculations, this approach, by reducing the size of each instance, makes it feasible to run each partial input completely in-core, thus allowing us to ignore disk access issues.

The quality of the tours generated by this approach will depend not just on the number k of pieces, but also how the partition is constructed. Ideally one would like cities that are close to each other to be in the same sets of the partition. For geometric instances, a standard approach is to partition the space in which the cities are located in contiguous regions (rectangles in the 2-dimensional case), This can be done efficiently and can be quite effective, leading to only a slight worsening of overall tour quality as k increases. There is unfortunately no obvious way to similarly exploit the geometry of high-dimensional Hamming-space instances. What structure the instances have that might be exploitable is implicit rather than explicit.

That such structure exists can be seen from experiments we report later for the **POWER** instance, where simply partitioning the instance based on the input order of the cities (the first n/k going into the first set, etc.) yields substantially better results than a random partition. This is because the instance is based on a visualization application, and was constructed based on walkthroughs of the model and thus the initial ordering of the cities to a certain extent reflects the implicit geometry of that model.

In general, however, we may not be given an instance in a form that already reflects such implicit structure, so it would be useful to have a generic partitioning procedure that can find at least some of the structure when it exists, and can run relatively quickly even when the instance does not fit in memory. To that end, we have devised the following scheme, which has enabled us essentially to match the effectiveness of the "original order" partition. Other methods are possible, but a full study of the alternatives is beyond the scope of this paper.

Our approach reduces the problem to one of finding a good ordering of the cities and then, as above, partitioning into k contiguous blocks of size roughly n/k. This reordering problem is then in turn solved by a combination of clustering and the solution of another (much smaller) TSP. (Using classical clustering algorithms to generate a partition directly doesn't work because those algorithms can yield clusters of widely varying size.) The overall schema of our approach looks like this:

1. Compute K *centers* from the input (K can be bigger than k).
2. Determine an order among the centers (a TSP).
3. For each center, reorder points in the associated cluster to form a tour, taking into account the identity of the clusters on either side.
4. Concatenate the tours.

Step 1 can be performed in one pass over the input, as can Step 3. Step 4 can be performed either via a sorting phase, or (if K is small) by a grouping algorithm that employs K passes.

Step 1: Computing the centers

Most clustering algorithms are designed to work on points in ℓ_2 spaces, rather than Hamming spaces. In addition, the size of our instances requires the use of *streaming* clustering methods. Various methods are possible, such as BIRCH [24] or the streaming K-median algorithm of Guha *et al.* [16]. We take the following related approach that computes K-centers:

Pick a uniform random sample of K points from the input data (all subsets of K points are equally likely). Then assign each point p to the sample point $s(p)$ it is closest to. Also determine p's l closest neighbors, and construct a bit vector v_p of length p where $v_p[i] = 1$ if

center $i \neq s(p)$ is one of p's l closest neighbors. Later, we will also use $\tilde{v}_p = v_p/\|v_p\|_1$, where $\|v\|_r$ denotes the ℓ_r-norm $\|v\|_r = (\sum_i v_i^r)^{1/r}$. We will abuse notation and use s to refer both to a sample point and the cluster of all points assigned to it. For each center s, let $v_s = \sum_{i \in s} v_p$. Normalize v_s so that $\|v_s\|_1 = 1 - \alpha$, and set $v_s[s] = \alpha$.

Intuitively, the vector v_s is a *signature* of this cluster, representing how other clusters appear when viewed from s. If two clusters are close together, then they will "see" all other clusters the same way, and so should be placed close to each other. Notice that if we dropped the restriction $i \neq s(p)$, then a large cluster could swamp v_s when normalized, rendering the neighbourhood information useless. The parameter α is a way of compromising, by fixing a contribution to this signature from s itself, but not making it too large.

In our implementation, we fix $l = 5$ and $\alpha = 0.2$ arbitrarily. The parameter K is set to 100.

Step 2: Determining an order among centers

Define $d(s_1, s_2) = \|v_{s_1} - v_{s_2}\|_1$. Compute a tour using this metric. For this we use the best of multiple runs of Iterated Lin-Kernighan (a variant of Lin-Kernighan [17]). .

Step 3: Reordering points in a cluster

Let s_l, s, s_r be three adjacent points in the tour generated in Step 2. Consider a point $p \in s$. If p's second nearest neighbour is s_l, then we place p in the set $L(s)$. If p's second nearest neighbour is s_r, then we place p in set $R(s)$. If neither case holds, we place p in set $L(s)$ or $M(s)$ depending on which of $d(\tilde{v}_p, v_{s_l})$ and $d(\tilde{v}_p, v_{s_r})$ is smaller.

All points in $L(s)$ are then sorted with respect to the measure $d(\tilde{v}_p, v_{s_l})$, and all points in $R(s)$ are sorted with respect to $d(\tilde{v}_p, v_{s_r})$. The output order for cluster s is then $L(s) \cdot s \cdot R(s)$.

The intuition here is that points more similar to s_l should appear closer in the order to s_l than to s_r and vice versa.

4 Sampling to Reduce Instance Size

One drawback of any partitioning scheme is that even if we divide into parts that can fit in main memory, it still may be the case that N is too large for us to do the required nearest neighbor computation or even simply to construct a true **NN** tour. Thus we will either have to partition the problem further, incurring additional penalties in tour length, or constrain our codes in some other way.

Since the basic issue is that there are too many potential distances that need to be computed, an appealing approach would be to discard a large proportion of the $N(N-1)/2$ edges as unimportant, and only concentrate on a sample of KN supposedly "good" edges. Our implementations of NN, 2-OPT, and 3-OPT can all take a "sparse graph" input consisting of a list of (edge,true distance) pairs and a default length for the distance between any pair of cities not represented in the list. They then find good tours with respect to this revised distance metric, which, depending on the quality of the sparse graph, will be fairly good tours for the original instance. The default length we typically use is $1,000,000$, meaning that the algorithms will work hard to use few "non-edges" and, subject to that, use as short a tour as possible.

Note that this approach introduces a second tradeoff: for a fixed memory size, choosing a smaller value of N means we have room for more edges per city, and hence possibly better tours. Tour quality will also depend on the quality of the sparse graph we construct. We obviously would like the edges included in this graph to connect relatively near cities.

A standard approach for computing such "near neighbors" in large high-dimensional data sets is to perform an embedding into a "nice" space like ℓ_2 and then reduce the dimension via projections or other means. The resulting set of points will approximately preserve the distances in the original set, and near neighbor calculations can then be performed efficiently.

There are a few problems with this approach. Hamming spaces are much harder to approximate than Euclidean spaces. Kushilevitz, Ostrovsky and Rabani present an algorithm [19] for computing approximate nearest neighbors in Hamming spaces; their algorithm requires space that is polynomial in n and d, and is thus impractical. Also, a result of Brinkman and Charikar [5] suggests that like ℓ_1, the Hamming metric is not amenable to embedding in ℓ_2 without undergoing severe distortion. Furthermore, as noted in [3], dimensionality reduction and the associated distortion in distance measurements are useful only if distances are *well-separated*; informally, the difference between a point's nearest neighbor and its farthest neighbor is large. For high-dimensional data sets, this is often not the case, and Table 4 demonstrates this for three example data sets that we use.

To resolve this issue, we exploit the "distance compression" that causes such a hindrance to approximate methods. Define $B_m(p)$ to be the ball of radius r, where r is m times the distance from p to its nearest neighbor. Assume there exists an $1 > \alpha > 0$ such that for any point p, an α-fraction of the points in the space lie in the ball $B_2(p)$. Intuitively, the larger α is, the less separated points are.

Fix a point p. If we now sample $1/\alpha$ points from P, with a constant probability one of these points lies in $B(p, r)$. The probability of success can be amplified further by sampling $\log n/\alpha$ points instead. This can be repeated, and so if we wish to compute k near neighbors (points that are at most twice the nearest

neighbour distance away from p), we merely sample $k \log n/\alpha$ points and take k points from this sample closest to p. The algorithm we use is sketched below. We use c as a boosting parameter to increase the probability of finding true near neighbors. The algorithm runs in two passes over the input.

Input: Set of points P, distance metric d, and parameters K, $\alpha > 0$, $c > 1$.
Output: For each point $p \in P$, set of K near neighbors.

for each p in a scan over P **do**
  Sample uniformly a set S of $s = cK$ points.
  Find K closest points in S to p and output p and this set.
end for

Algorithm 1: Algorithm

Note that under this scheme we sample c potential neighbors for each near neighbor we want to retain in the end. This preserves a certain standard of quality for the neighbor set as the number K of neighbors increases. It also keeps the time to construct a sparse subproblem fixed independent of the number k of such subproblems. However, it also means that the total time to construct all k subproblems in a k-way partition grows linearly with k, adding another tradeoff to our mix. We will discuss other options when we cover directions for further research at the end of the paper. In practice, we use a factor $c = 16$.

5 Related Work

The most well-known example of using reordering to improve compression is the Burrows-Wheeler transform [8]. Buchsbaum *et al.* [6] use the idea of column reordering to improve table compression. In a separate paper, Buchsbaum *et al.* [7] also use reorderings based on computing TSPs. Their TSP's also had an expensive-to-compute distance function – the distance between two columns reflects the improvement in compression achieved by `gzip` or another compression routine when the two columns were compressed together rather than separately. The TSP's arising in their application were however much smaller than ours (less than 1,000 cities), and so did not raise the running time and memory constraint issues that are our chief concern here.

The connection between TSP's and the number of runs was also made by Alizadeh*et al.* [2] in the context of reconstructing DNA sequences from probes, although their tests involved even smaller instances ($N = 100$).

Variants of the consecutive-ones property have been used in data mining [15] to identify interesting patterns in market-basket data. There is a vast body of research into mining based on association rules represented in terms of a boolean matrix; an extensive survey is beyond the scope of this paper. The paper by Cohen *et al.* [11] examines the problem of computing approximate near neighbors in a large Hamming space under a different measure (the so-called 'intersection over union' metric).

Managing large amounts of disk-resident data efficiently for real-time walkthroughs of three-dimensional geometric and radiositized databases is a well studied problem in computer graphics [13, 23, 1, 22]. Some preliminary work on reordering was done by Chhugani *et al.* [9]. That paper mentions the TSP heuristics we describe here, but focuses mainly on the data management and visualization aspects of the large-scale walkthrough problem.

Cortes *et al.* [12] proposed and developed the use of graph structures to determine *communities of interest* in order to detect patterns of calls among groups of customers; such patterns are also of interest in detecting fraud in telephone service.

6 Experiments

We now present a detailed experimental evaluation of our reordering strategy. We start with the experimental framework.

Data

POWER: Rows consist of transitions between visibility regions and columns are objects. An entry in the matrix is 1 if the object is in one of the regions defined by the transition and not in the other. The data comes from a 3D model of a power plant used for the walkthrough system developed by Chhugani *et al.* [9].

CITY: A similar data set obtained from a 3D model of the city of Atlanta.

PHONE: This data set consists of a sampling of call data from a major telecomunications company; rows correspond to callers and columns to callees (all data is anonymized).

Table 3 summarizes the basic properties of each data set.

	Rows	Columns	Sparsity (avg. entries/row)
POWER	47064	385828	8938
CITY	554455	789502	2000
PHONE	470728	543549	2

Table 3: Statistics for the data sets.

Data Characteristics

We demonstrate the *distance compression* that we discussed in Section 4. We sampled points at random

from the three data sets and computing their complete list of distances to other points in the set, and then calculated the size of $B_2(p)$ and $B_3(p)$ as a fraction of the number of points in the site. In Table 4, we record the average values.

	$B_2(p)$	$B_3(p)$
POWER	13.5%	24.6%
CITY	27.8%	36.2%
PHONE	35.4%	62.4%

Table 4: Aggregate Data Distribution statistics: For each data set, the first column indicates the fraction of points (on average) that lie in a ball of radius twice the nearest neighbour distance for a given point; the second column does the same for three times this value.

Platform

All experiments were run on a 32 CPU SGI with R12000 400 Mhz processors and 28 GB of main memory. The code for the sampling was written in `C` and compiled using `CC/gcc`. Auxiliary scripts for managing the data were written in `ksh/perl/awk`. For computing tours, we used the Johnson-McGeoch implementation from their survey of local search methods for the TSP [17].

Validation

The primary measure of quality we will use is *runs*(M), the number of runs on the reordered matrix. We will report the cost reduction achieved as a fraction of the identity ordering.

As mentioned earlier, a secondary benefit of reordering is the improvement in compression it yields (because long runs of 1s get placed together). To test this claim, we will use the well known *ExpGol* [21] compression scheme, studied by Johnson [18] in his 1999 review of methods for compressing bitmaps. We will use the codecs that he designed. For more details on how *ExpGol* works, we refer the reader to the paper by Moffat and Zobel [21].

Compression schemes like *ExpGol* are designed to compress sequences of integers, and work well when the integers are small. In a sparse representation, each row of the boolean matrix is a sequence of column IDs; however these are not small. Consider such a sequence $x_1, x_2, \ldots x_n$. *Offset mapping* (*Offset*) will convert this sequence to the form $x_1, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}$. A run of k 1s starting at column ID x in the matrix will thus be represented as $(x, 1, 1, 1, ...$k-1 times$)$, A more efficient method is the well known *run-length encoding* that would represent the run by the pair (x, k).

A slightly better method called *two-sided run length encoding* (*2SidedRLE*) exploits the fact that runs of 0s and 1s can be encoded simultaneously. Each sequence of integers is represented as the sequence $\mathbf{0}_1, \mathbf{1}_1, \mathbf{0}_2, \ldots$, where $\mathbf{0}_i$ is the length of the i^{th} sequence of 0s and $\mathbf{1}_j$ is the length of the j^{th} sequence of 1s[1]. The advantage of this approach is that if intervals of 1s and 0s are interspersed, then there are no large numbers in the resulting sequence.

We will evaluate our algorithms by first using *Offset* or *2SidedRLE* to transform each row, and then using *ExpGol* to compress the row. The total compressed size will be the sum of compressed sizes for each row. It will often happen that the best compression scheme for the original input is different from the best scheme after reordering. In all cases, we use the best results to determine compression ratios.

Table 5 summarizes the cost of the input data under these three measures.

	runs(M)	*Offset* +*ExpGol*	*2SidedRLE* +*ExpGol*
POWER	251689169	235460917	245255311
CITY	807562396	823185832	906690088
PHONE	826766	4029710	4665946

Table 5: Three different cost measures for the data.

Reporting: In all the results below, we report the *relative improvement* achieved with respect to the identity ordering. For many situations, we will have tours generated using sampling to compute smaller instances, and tours generated from a full instance (typically only when the pieces are small). We will use the term *sample* to refer to tours generated by sampling, and *full* to refer to tours generated by exact distance computations. All times are reported in seconds.

6.1 Overall performance of the algorithm

We start with a start-to-finish evaluation of our schemes on the three data sets. Results reported in this section reflect the best choice of parameters at various stages in the computation pipeline, independent of running time; the goal is to verify that the *null hypothesis*, that reordering has no effect, is false.

Table 6 summarizes the results achieved.

	runs(M)	Compression
POWER	58%	38%
CITY	34%	25%
PHONE	35%	15%

Table 6: Relative improvement of **REORDER** with respect to the identity ordering.

In all cases, we get a substantial improvement after reordering the data. It is worth mentioning that in the case of **PHONE**, the compression achieved is small (we reduce the input by 15%) because the data set itself is extremely sparse, having an average of two entries per row.

[1]This scheme can be interpreted as performing run length encoding, and then doing further run length encoding on the start IDs of each tuple.

For all the data sets, the best approach (independent of time) is obtained by splitting the data set into the fewest number of subproblems that can individually fit in memory, and then computing full tours i.e tours based on looking at the entire subproblem. In general, the best tours are obtained using **3-OPT**, although there are occasional situations where this is not true.

For **POWER**, the best split comes at $k = 16$, and for city at $k = 32$. The data set **PHONE** is small enough to fit entirely in memory.

6.2 A Closer Look At The Approach

We now undertake a more detailed study of how various parameter choices in our reordering strategy affect the running time and the resulting cost.

6.2.1 Number Of Subproblems And Sampling

The number of subproblems the input is split into prior to tour computation is constrained by the TSP algorithm. In order to run in-core, it needs to be able to fit either the sampled sparse graph or the entire subproblem into main memory. Thus, the number of neighbours computed for each point in a subproblem will be inversely related to the size of the piece.

We first illustrate how the performance of the tours varies with the number of pieces. Table 8 illustrates the improvement in *runs*(M) achieved, in all cases using **3-OPT** as the tour construction algorithm. Notice that in all cases, sampling yields better results as the number of subproblems increases, because this allows us to sample more "edges" in the sparse graph for each input point.

		Number of Subproblems (k)			
		4	8	16	32
POWER	*sample*	21%	35%	48%	54%
POWER	*full*	–	–	57%	–
CITY	*sample*	–	7%	16%	26%
CITY	*full*	–	–	–	34%
PHONE	*sample*	1%	2%	3%	–
PHONE	*full*	18%	14%	11%	–

Table 8: Relative Improvement in *runs*(M) as number of subproblems increases. For each data set, a *full* tour was computed for the value of k such that each subproblem fits in memory.

However, partitioning and computing a tour based on the full instance is always better. Table 8 also illustrates that it is important to partition into the *fewest* number of subproblems that allow each subproblem to fit in memory. In the case of **PHONE**, where the entire input fits in memory, partitioning only degrades the performance.

When time is no object, the best strategy is clearly to compute *full* tours on as few subproblems as possible. However, this is significantly slower than sampling, because of the quadratic distance computations. Table 7 compares running times of sampling and full instance-based approaches; notice that once sampling is performed, the TSP cost itself is very small.

6.2.2 Tour Algorithms

We use three algorithms to compute tours; **NN**, **2-OPT** and **3-OPT**. As we described in Section 3, **2-OPT** and **3-OPT** perform local refinements on a tour generated by **NN**, so in general for full instances **3-OPT** achieves better compression than **2-OPT**, which in turns improves upon **NN**. However there is a tradeoff between the time spent optimizing, and the improvement obtained. Table 9 summarizes the relative improvement (and the time needed for these improvements) for the three algorithms. The numbers reflect the best-case settings for the other parameters, and only show time for computing the TSP.

What the table indicates is that unlike the case of Euclidean TSPs, even an **NN**-based tour is a good approximation to the best answer we can achieve. The times reported here for **NN** are overestimates, because they include time spent constructing neighbour lists for the **2-OPT** and **3-OPT** phases of the tour construction.

Interestingly, when we construct tours from sampled instances, it is not always the case that **2-OPT** and **3-OPT** are better than **NN**. For reasons discussed in Section 4, the tour construction algorithm have to deal with the occasional "infinite" edge in the sparse graph. As a result, **NN** can sometimes give the best cost solution. One such example is the 8-way partition for **CITY**. In this instance, the improvement achieved by **NN** is 7% in comparison to 5% achieved by **2-OPT** and **3-OPT**

6.3 Finding A Good Starting Order

Finding a good starting order can be a crucial component of a successful reordering. We start with Table 10, comparing the cost of a random reordering to the cost achieved by **REORDER**. Once again, we use best-case settings for all the data sets.

	Random	Identity	**REORDER**
POWER	+52%	–	-58%
PHONE	+0.4%	–	-35%
CITY	+36%	–	-34%

Table 10: Why the initial ordering matters: Random orderings are bad.

This table suggests that the initial orderings given to us in the case of **POWER** and **PHONE** were quite good. For **PHONE**, the initial ordering is close to random. This is not a coincidence; there is strong spatial structure in the way the elements of **POWER** and

	POWER				CITY				PHONE			
	sampling			*full*	*sampling*			*full*	*sampling*			*full*
	Sampling	TSP	Total	–	Sampling	TSP	Total		Sampling	TSP	Total	
4	37429	3736	41165	–	–	–	–		2723	4903	7626	11802
8	106748	3618	110366	–	112020	6166	118186		5644	4482	10126	5606
16	149388	2842	152230	202252	216159	5676	221835		5287	3585	8872	2925
32	330719	3792	334511	–	375940	6089	382029	823052	–	–	–	

Table 7: Time taken (in seconds) by **REORDER** as the number of tour pieces varies

	POWER		PHONE		CITY	
	Time	*runs*(M)	Time	*runs*(M)	Time	*runs*(M)
NN	192520	55.8%	43308	30.4%	772745	32.6%
2-OPT	192878	57.0%	–	–	774292	33.5%
3-OPT	202252	57.4%	43688	35.2%	823052	34.1%

Table 9: Relative performance of the three TSP heuristics (in seconds). The running times do not include sampling time. The times measured are cumulative, since **3-OPT** takes the output of **NN** as its input. We did not measure the **2-OPT** time for **PHONE** because running **3-OPT** (with its better quality solution) takes only a few seconds more than running **2-OPT**.

PHONE were generated. Note that since **PHONE** can be run entirely in main memory and thus does not need to be partitioned, no prior reordering will have an effect on the outcome. Thus, in the sequel, we only present results for **POWER** and **CITY**.

Our next experiment checks the overall efficacy of clustering alone. We evaluate the quality of the reordering computed using the clustering strategy of Section 3.1. Since our clustering strategy is independent of the input, all comparisons will be made with respect to a random ordering. Observe that the reordered input is slightly better than the original input we were given.

	Identity	Clustering
POWER	34%	43 %
CITY	26%	27 %

Table 11: The improvement in quality of the ordering generated by clustering, relative to a random ordering

Finally, we use the new order generated via clustering to partition the input as before, and compute tours in the manner described above. Once again, we present results for the optimal combination of settings. Table 12 presents these results, in comparison to the results obtained without clustering for the same parameter settings.

	Reordering based on given order	Reordering based on clustering
POWER	72.0%	71.1%
CITY	51.7%	51.4%

Table 12: Evaluation of improvement due to clustering and reordering phase, relative to a random ordering

It is important to note that the clustering strategy is *oblivious of the input ordering*. In other words, merely by clustering the data, we were able to recreate (mostly) all the locality-preserving properties of the original input. Thus, for an arbitrary input set, an initial clustering phase to generate an initial order can greatly improve the effectiveness of tour computation in the next phase of the reordering process.

As a final experiment, we randomly reordered **POWER** and used this random ordering as the starting order for **REORDER**. The run reduction we obtain, again using the best possible settings of a 16-way split and **3-OPT**, is 57.4%, which is less than the 71.1% improvement obtained using clustering. This further demonstrates the value of a good starting order.

6.4 Discussion

We conclude this section with a review of the major findings. Overall, the use of a TSP for reordering yields significant benefits, both in terms of access cost (measured by *runs*(M)), and in terms of compression. If the input is too large to fit entirely in main memory, an effective strategy is to break it into pieces small enough to fit in memory, and compute tours for each piece, using **3-OPT**.

If time is a constraint, and memory is limited, sampling is a good way to generate tours that are reasonably good in a small amount of time. Again, the best strategy is to use smaller pieces and pick more neighbours.

The improvement achieved via partitioning and/or sampling is a function of how well-ordered the original input is. The better the input ordering, the better the output. Therefore an initial clustering step that tries to group points in clusters if they are near each other is very effective in creating a good starting order to partition.

The type of algorithm used to compute the tour matters somewhat less than has been traditionally observed for TSPs. **NN**-based tours are close to, but worse than tours based on **2-OPT** and **3-OPT**, and so a choice of which method to use can be based on the amount of time one is willing to spend.

7 Directions for Further Research

As in most papers concerned with how to handle out-of-memory problems, we have concentrated here on just a few instances and applications. In future work we hope to confirm the wider applicability of our approach.

For example, much larger instances from the **PHONE** application exist and would provide new challenges – the total number of phone-numbers is in the 100's of millions, not 100's of thousands as studied here. We would also like to study instancs from "association rule" applications, for example data sets in which the rows correspond to documents and the columns to key words (or vice versa).

In addition, several algorithmic questions are worthy of further study. One of our conclusions here was that the best results were obtained by partitioning down to the largest subproblems for which it was feasible to run 3-OPT on the full subproblems and then doing so. Is this true in general? If we need to partition into 1024 subproblems to get them small enough for 3-OPT, might not the subdivision penalty be so severe that it would be better to solve sparse versions of the subproblems in a 32-way partition?

Another question has to do with the quality of the sampled subproblems. Here we allowed a fixed sampling time to create a subproblem, independent of its size. What if we only allowed a fixed amount of time overall to create all the sparse subproblems? This would mean even though the sparse subproblems have more edges when k is large, they would be of lower average quality, so the improvement in results as k grows might be lessened (or disappear).

Finally, for particular applications, are there application-specific sampling techniques that might generate better sparse graphs? For example, in the **PHONE** application, one might exploit the "community of interest" ideas of Cortes *et al.* [12], and get neighbors for a given column by preferentially sampling columns corresponding to phone numbers that call or are called by the given column's number, and columns that correspond to numbers that *those* numbers call.

8 Acknowledgements

We thank Ted Johnson for providing us with his code for *ExpGol* compression and Lyle McGeoch for modifications to the TSP codes to handle Hamming metrics. We also would like to thank Divesh Srivastava for useful discussions and pointers to related work.

References

[1] Aliaga, D., Cohen, J., Wilson, A., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Baker, E., Bastos, R., Whitton, M., Brooks, F., and Manocha, D. Mmr: An integrated massive model rendering system using geometric and image-based acceleration. In *Proc. of ACM Symposium on Interactive 3D Graphics* (1999), pp. 199–206.

[2] Alizadeh, F., Karp, R. M., Newberg, L. A., and Weisser, D. K. Physical mapping of chromosomes: A combinatorial problem in molecular biology. In *Proc. 3rd ACM-SIAM Symp. Discrete Algorithms* (1993), pp. 371–381.

[3] Beyer, K., Goldstein, J., Ramakrishnan, R., and Shaft, U. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science 1540* (1999), 217–235.

[4] Booth, K. S., and Lueker, G. S. Testing for the consecutive ones property, interval graphs, and graph planarity using P-Q tree algorithms. *J. of Comp. and Syst. Sci. 13* (1976), 335–379.

[5] Brinkman, B., and Charikar, M. On the impossibility of dimension reduction in l_1. In *Proc. 44th IEEE Symp. Foundations of Computer Science* (2003), pp. 514–523.

[6] Buchsbaum, A. L., Caldwell, D. F., Church, K. W., Fowler, G. S., and Muthukrishnan, S. Engineering the compression of massive tables: an experimental approach. In *Proc. 10th ACM-SIAM Symp. Discrete Algorithms* (2000), Society for Industrial and Applied Mathematics, pp. 175–184.

[7] Buchsbaum, A. L., Fowler, G. S., and Giancarlo, R. Improving table compression with combinatorial optimization. *J. ACM 50*, 6 (2003), 825–851.

[8] Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression system. Tech. Rep. 124, DEC SRC, 1994.

[9] Chhugani, J., Purnomo, B., Krishnan, S., Cohen, J., Venkatasubramanian, S., Johnson, D., and Kumar, S. vLOD: High-fidelity walkthrough of large virtual environments. Submitted., 2003.

[10] Christofides, N. Worst-case analysis of a new heuristic for the travelling salesman problem. Tech. Rep. 388, Graduate School of Industrial Administration, CMU, 1976.

[11] Cohen, E., Datar, M., Fujiwara, S., Gionis, A., Indyk, P., Motwani, R., Ullman, J. D., and Yang, C. Finding interesting associations without support pruning. *Knowledge and Data Engineering 13*, 1 (2001), 64–78.

[12] Cortes, C., Pregibon, D., and Volinsky, C. Computational methods for dynamic graphs. *Journal of Computational and Graphical Statistics 12* (2003), 950–970.

[13] Funkhouser, T. A., Sequin, C. H., and Teller, S. J. Management of large amounts of data in interactive building walkthroughs. In *Computer Graphics (1992 Symposium on Interactive 3D Graphics)* (Mar. 1992), D. Zeltzer, Ed., vol. 25, pp. 11–20.

[14] Garey, M. R., and Johnson, D. S. *Computers and Intractability.* W. H. Freeman, 1979.

[15] Gionis, A., Kujala, T., and Mannila, H. Fragments of order. In *Proc. 9th ACM Conf. Knowledge Discovery and Data Mining* (2003).

[16] Guha, S., Meyerson, A., Mishra, N., Motwani, R., and O'Callaghan, L. Clustering data streams: Theory and practice. *IEEE Trans. Knowl. Data Eng 15*, 3 (2003), 515–528.

[17] Johnson, D. S., and McGeoch, L. A. *The Traveling Salesman Problem: A Case Study in Local Optimization.* John Wiley and Sons, 1997, ch. 8.

[18] Johnson, T. Performance measurements of compressed bitmap indices. In *Proc. 25th Intnl. Conf. Very Large Databases (VLDB)* (1999).

[19] Kushilevitz, E., Ostrovsky, R., and Rabani, Y. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proc. 30th ACM Symp. Theory of Computing* (1998), pp. 614–623.

[20] Lin, S., and Kernighan, B. An effective heuristic algorithm for the traveling-salesman proble m. *Operations Research 21* (1973), 498–516.

[21] Moffat, A., and Zobel, J. Parameterised compression for sparse bitmaps. In *Research and Development in Information Retrieval* (1992), pp. 274–285.

[22] Shou, L., Chionh, J., Huang, Z., Ruan, Y., and Tan, K.-L. Walking through a very large virtual environment in real-time. In *Proc. 27th Intnl. Conf. Very Large Databases (VLDB)* (2001).

[23] Teller, S., Fowler, C., Funkhouser, T., and Hanrahan, P. Partitioning and ordering large radiosity computations. In *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)* (July 1994), A. Glassner, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, ACM Press, pp. 443–450. ISBN 0-89791-667-0.

[24] Zhang, T., Ramakrishnan, R., and Livny, M. Birch: an efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (1996), ACM Press, pp. 103–114.