Mining for Empty Rectangles in Large Data Sets

Jeff Edmonds[1], Jarek Gryz[1], Dongming Liang[1], and Renée J. Miller[2]

[1] York University
[2] University of Toronto

Abstract. Many data mining approaches focus on the discovery of similar (and frequent) data values in large data sets. We present an alternative, but complementary approach in which we search for empty regions in the data. We consider the problem of finding all maximal empty rectangles in large, two-dimensional data sets. We introduce a novel, scalable algorithm for finding all such rectangles. The algorithm achieves this with a single scan over a sorted data set and requires only a small bounded amount of memory. We extend the algorithm to find all maximal empty hyper-rectangles in a multi-dimensional space. We consider the complexity of this search problem and present new bounds on the number of maximal empty hyper-rectangles. We briefly overview experimental results obtained by applying our algorithm to real and synthetic data sets and describe one application of empty-space knowledge to query optimization.

1 Introduction

Much work in data mining has focused on characterizing the similarity of data values in large data sets. This work includes clustering or classification in which different techniques are used to group and characterize the data. Such techniques permit the development of more "parsimonious" versions of the data. Parsimony may be measured by the degree of compression (size reduction) between the original data and its mined characterization [3]. Parsimony may also be measured by the semantic value of the characterization in revealing hidden patterns and trends in the data [8, 1].

Consider the data of Figure 1 representing information about traffic infractions (tickets), vehicle registrations, and drivers. Using association rules, one may discover that Officer Seth gave out mostly speeding tickets [1] or that drivers of BMWs usually get speeding tickets over $100 [11]. Using clustering one may discover that many expensive (over $500) speeding tickets were given out to drivers of BMW's [14]. Using fascicles, one may discover that officers Seth, Murray and Jones gave out tickets for similar amounts on similar days [8].

The data patterns discovered by these techniques are defined by some measure of similarity (data values must be identical or similar to appear together in a pattern) and some measure of degree of frequency or occurrence (a pattern is only interesting if a sufficient number of data values manifest the pattern or, in the case of outlier detection, if very few values manifest the pattern).

Registration

RegNum	*Model*	*Owner*
R43999	Saab9.5W	Owen
R44000	HondaCW	Wang
....		

Tickets

Tid	*Officer*	*RegNum*	*Date*	*Infraction*	*Amt*	*DLNum*
119	Seth	R43999	1/1/99	Speed	100	G4337
249	Murray	R00222	2/2/95	Parking	30	G7123
...						

Drivers

DLNum	*Name*	*DOB*
G16999	Smith	1970-03-22
G65000	Simon	1908-03-05
....		

Fig. 1. Schema and Data of a Traffic Infraction Database

In this paper, we propose an alternative, but complementary approach to characterizing data. Specifically, we focus on finding and characterizing empty regions in the data. In the above data set, we would like to discover if there are certain ranges of the attributes that never appear together. For example, it may be the case that no tickets were issued to BMW Z3 series cars before 1997 or that no tickets for over $1,000 were issued before 1990 or that there is no record of any tickets issued after 1990 for drivers born before 1920. Some of these empty regions may be foreseeable (perhaps BMW Z3 series cars were first produced in 1997). Others may have more complex or uncertain causes (perhaps older drivers tend to drive less and more defensively).

Clearly, knowledge of empty regions may be valuable in and of itself as it may reveal unknown correlations between data values which can be exploited in applications.[1] For example, if a DBA determines that a certain empty region is a time invariant constraint, then it may be modeled as an integrity constraint. Knowing that no tickets for over $1,000 were issued before 1990, a DBA of a relational DBMS can add a check constraint to the Tickets table. Such constraints have been exploited in semantic query optimization [5].

To maximize the use of empty space knowledge, our goal in this work is to not only find empty regions in the data, but to fully characterize that empty space. Specifically, we discover the set of all maximal empty rectangles. In Section 2, we formally introduce this problem and place our work in the context of related work from the computational geometry and artificial intelligence communities. In Section 3, we present an algorithm for finding the set of all maximal empty rectangles in a two-dimensional data set. Unlike previous work in this area, we focus on providing an algorithm that scales well to large data sets. Our algorithm requires a single scan of a sorted data set and uses a small, bounded amount of memory to compute the set of all maximal empty rectangles. In contrast, related algorithms require space that is at least on the order of the size of the data set. We extend the algorithm to multiple dimensions and present complexity results along with bounds on the number of maximal hyper-rectangles. In Section 4, we present the results of experiments performed on both synthetic and real data showing the scalability of our mining algorithm. We also consider the nature and quantity of empty rectangles that can occur in large, real databases. We conclude in Section 5.

[1] [10] describe applications of such correlations in a medical domain.

2 Problem Definition and Related Work

Consider a data set D consisting of a set of tuples $\langle v_x, v_y \rangle$ over two totally ordered domains. Let X and Y denote the set of distinct values in the data set in each of the dimensions. We can depict the data set as an $|X| \times |Y|$ matrix M of 0's and 1's. There is a 1 in position $\langle x, y \rangle$ of the matrix if and only if $\langle v_x, v_y \rangle \in D$ where v_x is the x^{th} smallest value in X and v_y the y^{th} smallest in Y.

An empty rectangle is *maximal* [2] if it cannot be extended along either the X or Y axis because there is at least one 1-entry on each of the borders of the rectangle. Although it appears that there may be a huge number of overlapping maximal rectangles, [12] proves that the number is at most $\mathcal{O}(|D|^2)$, and that for a random placement of the 1-entries the expected value is $\mathcal{O}(|D| \log |D|)$ [12]. We prove that the number is at most $\mathcal{O}(|X||Y|)$ (Theorem 4).

A related problem attempts to find the minimum number of rectangles (either overlapping or not) that covers all the 0's in the matrix. This problem is a special case of the problem known as Rectilinear Picture Compression and is NP-complete [7]. Hence, it is impractical for use in large data sets.

The problem of finding empty rectangles or hyper-rectangles has been studied in both the machine learning [10] and computational geometry literature [12, 2, 4, 13]. Liu *et al* motivate the use of empty space knowledge for discovering constraints (in their terms, *impossible combinations of values*) [10]. However, the proposed algorithm is memory-based and not optimized for large datasets. As the data is scanned, a data structure is kept storing all maximal hyper-rectangles. The algorithm runs in $\mathcal{O}(|D|^{2(d-1)} d^3 (log\ |D|)^2)$ where d is the number of dimensions in the data set. Even in two dimensions ($d = 2$) this algorithm is impractical for large datasets. In an attempt to address both the time and space complexity, the authors propose only maintaining maximal empty hyper-rectangles that exceed an *a priori* set minimum size. This heuristic is only effective if this minimum size is set sufficiently small. Furthermore, as our experiments on real dataset will show, for a given size, there are typically many maximal empty rectangles that are largely overlapping. Hence, this heuristic may yield a set of large, but almost identical rectangles. This reduces the effectiveness of the algorithm for a large class of data mining applications where the number of discovered regions is less important that the distinctiveness of the regions. Other heuristic approaches have been proposed that use decision tree classifiers to (approximately) separate occupied from unoccupied space then post-process the discovered regions to determine maximal empty rectangles [9]. Unlike our approach, these heuristics do not guarantee that all maximal empty rectangles are found.

This problem has also been studied in the computational geometry literature [12, 2, 4, 13] where the primary goal has been to produce run time bounds. These algorithms find all maximal empty rectangles in time $\mathcal{O}(|D| \log |D| + s)$ and space $\mathcal{O}(|D|)$, where $|D|$ is the size of the data set and s denotes the number of maximal empty rectangles. Such algorithms are particularly effective if the

[2] Do not confuse maximal with maximum (largest).

data set is very sparse and there happens to be only a few maximal rectangles. However, these algorithms do not scale well for large data sets because of their space requirements. The algorithms must continually access and modify a data structure that is as large as the data set itself. Because in practice this will not fit in memory, an infeasible amount of disk access is required on large data sets.

The setting of the algorithm in [13] is different because it considers points in the real plane instead of 1-entries in a matrix. The only difference that this amounts to is that they assume that points have distinct X and Y coordinates. This is potentially a problem for a database application since it would not allow any duplicate values in data (along any dimension).

Despite the extensive literature on this problem, none of the known algorithms are effective for large data sets. Even for two-dimensional data sets, the only known technique for scaling these algorithms is to provide a fixed bound on the size of the empty rectangles discovered, a technique which severally limits the application of the discovered results.

Our first contribution to this problem is an algorithm for finding all maximal empty rectangles in a two-dimensional space that can perform efficiently in a bounded amount of memory and is scalable to a large, non-memory resident data set. Unlike the algorithm of [10], our algorithm requires the data be processed in sorted order. However, sorting is a highly optimized operation within modern DBMS and by taking advantage of existing scalable sorting techniques, we have produced an algorithm with running time $\mathcal{O}(|X||Y|)$ that requires only a single scan over the sorted data. Furthermore, the memory requirements are $\Theta(|X|)$, which is an order of magnitude smaller than the size $\mathcal{O}(|X||Y|)$ of both the input and the output. (We assume without loss of generality that $|X| \leq |Y|$.) If the memory available is not sufficient, our algorithm could be modified to run on a portion of the matrix at a time at the cost of extra scans of the data set. Our second main contribution is an extension of our algorithm to find all maximal empty hyper-rectangles in multi-dimensional data. The space and time trade-off compare favorably to those of the heuristic algorithm of [10] (the time complexity of our extended algorithm is $\mathcal{O}(d|D|^{2(d-1)})$ and the space requirements are $\mathcal{O}(d^2|D|^{d-1})$), but are worse than those of incomplete classifier-based algorithms [9].

3 Algorithm for Finding All Maximal Empty Rectangles

This section presents an elegant algorithm for finding all maximal empty regions within a two dimensional data set. Although the binary matrix M representation of the data set D is never actually constructed, for simplicity we describe the algorithm completely in terms of M. In doing so, however, we must insure that only one pass is made through the data set D.

The main structure of the algorithm is to consider each 0-entry $\langle x, y \rangle$ of M one at a time row by row. Although the 0-entries are not explicitly stored, this is simulated as follows. We assume that the set X of distinct values in the (smaller) dimension is small enough to store in memory. The data set D is stored on disk

sorted with respect to Y, X. Tuples from D will be read sequentially off the disk in this sorted order. When the next tuple $\langle v_x, v_y \rangle \in D$ is read from disk, we will be able to deduce the block of 0-entries in the row before this 1-entry.

When considering the 0-entry $\langle x, y \rangle$, the algorithm needs to look ahead by querying the matrix entries $\langle x+1, y \rangle$ and $\langle x, y+1 \rangle$. This is handled by having the single pass through the data set actually occur one row in advance. This extra row of the matrix is small enough to be stored in memory. Similarly, when considering the 0-entry $\langle x, y \rangle$, the algorithm will have to look back and query information about the parts of the matrix already read. To avoid re-reading the data set, all such information is retained in memory.

The main data structure maintained by the algorithm is the *maximal staircase*, $staircase(x, y)$, which stores the shape of the maximal staircase shaped block of 0-entries starting at entry $\langle x, y \rangle$ and extending up and to the left as far as possible. See Figure 2. Note that the bottom-right entry separating two steps of the staircase is a 1-entry. This entry prevents the two adjoining steps from extending up or to the left and prevents another step forming between them.

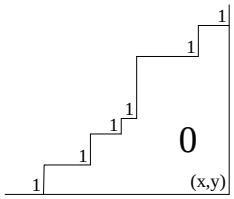

Fig. 2. The maximal staircase for $\langle x, y \rangle$.

```
loop  y = 1 ... n
  loop  x = 1 ... m
    (I) Construct staircase (x,y)
    (II) Output all maximal 0-rectangles with <x,y>
         as the bottom-right corner
```

Fig. 3. Algorithm Structure.

The purpose of constructing the $staircase(x, y)$ is to output all maximal rectangles that lie entirely *within* that staircase and whose bottom right corner is $\langle x, y \rangle$. The algorithm (Figure 3) traverses the matrix left-to-right and top-to-bottom creating a staircase for every entry in the matrix. We now describe the construction of the staircase and the production of maximal empty rectangles in detail.

3.1 Constructing $staircase(x, y)$

The maximal staircase, $staircase(x, y)$, is specified by the coordinates of the top-left corner $\langle x_i, y_i \rangle$ of each of its steps. This sequence of steps $\langle \langle x_1, y_1 \rangle, \ldots, \langle x_r, y_r \rangle \rangle$ is stored in a stack, with the top step $\langle x_r, y_r \rangle$ on the top of the stack.

The maximal staircase, $staircase(x, y) = \langle \langle x_1, y_1 \rangle, \ldots, \langle x_r, y_r \rangle \rangle$, is easily constructed from the staircase, $staircase(x-1, y) = \langle \langle x'_1, y'_1 \rangle, \ldots, \langle x'_{r'}, y'_{r'} \rangle \rangle$ as follows. See Figure 4. We start by computing y_r, which will be the Y-coordinate for the highest entry in $staircase(x, y)$. If the $\langle x, y \rangle$ entry itself is a 1, then $staircase(x, y)$ is empty. Otherwise, continue moving up through column x from $\langle x, y \rangle$ as long as the entry contains a 0. The entry y_r is the Y-coordinate of the

last 0-entry in column x above $\langle x, y\rangle$ before the first 1-entry is found. How the rest of $staircase(x, y)$ is constructed depends on how the new height of top step y_r compares with the old one $y'_{r'}$.

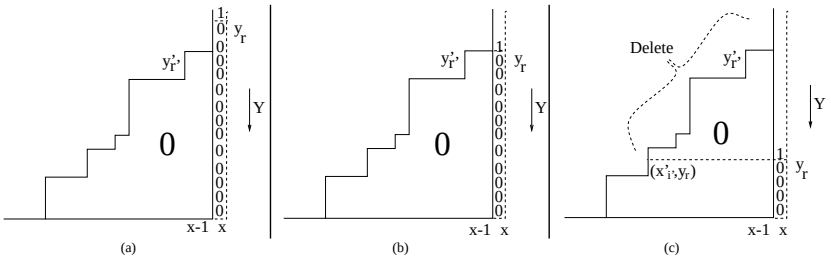

Fig. 4. The three cases in constructing maximal staircase, $staircase(x, y)$, from $staircase(x-1, y)$.

Case $y_r < y'_{r'}$**:** Figure 4(a). If the new top step is higher than the old top step, then the new staircase $staircase(x, y)$ is the same as the old one $staircase(x-1, y)$ except one extra high step is added on the right. This step will have width of only one column and its top-left corner will be $\langle x, y_r\rangle$. In this case, $staircase(x, y)$ is constructed from $staircase(x-1, y)$ simply by pushing this new step $\langle x, y_r\rangle$ onto the top of the stack.

Case $y_r = y'_{r'}$**:** Figure 4(b). If the new top step has the exact same height as the old top step, then the new staircase $staircase(x, y)$ is the same as the old one $staircase(x-1, y)$ except that this top step is extended one column to the right. Because the data structure $staircase(x, y)$ stores only the top-left corners of each step, no change to the data structure is required.

Case $y_r > y'_{r'}$**:** Figure 4(c). If the new top step is lower then the old top step, then all the old steps that are higher then this new highest step must be deleted. The last deleted step is replaced with the new highest step. The new highest step will have top edge at y_r and will extend to the left as far as the last step $\langle x'_{i'}, y'_{i'}\rangle$ to be deleted. Hence, top-left corner of this new top step will be at location $\langle x'_{i'}, y_r\rangle$. In this case, $staircase(x, y)$ is constructed from $staircase(x-1, y)$ simply by popping off the stack the steps $\langle x'_{r'}, y'_{r'}\rangle, \langle x'_{r'-1}, y'_{r'-1}\rangle, \ldots, \langle x'_{i'}, y'_{i'}\rangle$ as long as $y_r > y'_i$. Finally, the new top step $\langle x'_{i'}, y_r\rangle$ is pushed on top.

One key thing to note is that when constructing $staircase(x, y)$ from $staircase(x-1, y)$, *at most* one new step is created.

3.2 Outputting the Maximal 0-rectangles

The goal of the main loop is to output all maximal 0-rectangles with $\langle x, y\rangle$ as the bottom-right corner. This is done by outputting all steps of $staircase(x, y)$ that

cannot be extended down or to the right. Whether such a step can be extended depends on where the first 1-entry is located within row $y+1$ and where it is within column $x+1$. Consider the largest block of 0-entries in row $y+1$ starting at entry $\langle x, y+1\rangle$ and extending to the left. Let x_* be the X-coordinate of this left most 0-entry (see Figure 5). Similarly, consider the largest block of 0-entries in column $x+1$ starting at entry $\langle x+1, y\rangle$ and extending up. Let y_* be the Y-coordinate of this top most 0-entry. The following theorem states which of the rectangles within the $staircase(x, y)$ are maximal.

Theorem 1. *Consider a step in $staircase(x, y)$ with top-left corner $\langle x_i, y_i\rangle$. The rectangle $\langle x_i, x, y_i, y\rangle$ is maximal if and only if $x_i < x_*$ and $y_i < y_*$.*

Proof. The step $\langle x_i, y_i\rangle$ of $staircase(x, y)$ forms the rectangle $\langle x_i, x, y_i, y\rangle$. If $x_i \geq x_*$, then this rectangle is sufficiently skinny to be extended down into the block of 0-entries in row $y+1$. For example, the highest step in Figure 5 satisfies this condition. On the other hand, if $x_i < x_*$, then this rectangle cannot be extended down because it is blocked by the 1-entry located at $\langle x_*-1, y+1\rangle$. Similarly, the rectangle is sufficiently short to be extended to the right into the block of 0-entries in column $x+1$ only if $y_i \geq y_*$. See the lowest step in Figure 5. Hence, the rectangle is maximal if and only if $x_i < x_*$ and $y_i < y_*$.

To output the steps that are maximal 0-rectangles, pop the steps $\langle x_r, y_r\rangle$, $\langle x_{r-1}, y_{r-1}\rangle$, ... from the stack. The x_i values will get progressively smaller and the y_i values will get progressively larger. Hence, the steps can be divided into three intervals. At first, the steps may have the property $x_i \geq x_*$. As said, these steps are too skinny to be maximal. Eventually, the x_i of the steps will decrease until $x_i < x_*$. Then there may be an interval of steps for which $x_i < x_*$ and $y_i < y_*$. These steps are maximal. For these steps output the rectangle $\langle x_i, x, y_i, y\rangle$. However, y_i may continue to increase until $y_i \geq y_*$. The remaining steps will be too short to be maximal.

The staircase steps in the third interval do not need to be popped, that is, after $y_i \geq y_*$, because they are not maximal. These stay on the stack. Conveniently the steps from the first and second interval, that is, while $y_i < y_*$, can be thrown way as they are popped off the stack, because they are precisely the steps that are thrown away when constructing $staircase(x+1, y)$ from $staircase(x, y)$. (Recall that the next step after outputting the maximal steps in $staircase(x, y)$ is to construct $staircase(x+1, y)$ from $staircase(x, y)$.)

The reason these staircase steps are precisely the ones to be deleted is as follows. Note that the value y_* used to know which steps in $staircase(x, y)$ are maximal and the y_r used in the construction of $staircase(x+1, y)$ are both the Y-coordinate of the top most 0-entry in the block of 0-entries in column $x+1$ starting at entry $\langle x+1, y\rangle$ and extending up. Hence, y_* from $staircase(x, y)$ is the same as y_r for $staircase(x+1, y)$. It follows that in constructing $staircase(x+1, y)$, the steps in $stair(x, y)$ are popped and deleted as long as $y_i < y_* = y_r$ (Figure 4(c)).

3.3 Time and Space Complexity

Theorem 2. *The time complexity of the algorithm is $\mathcal{O}(nm)$.*

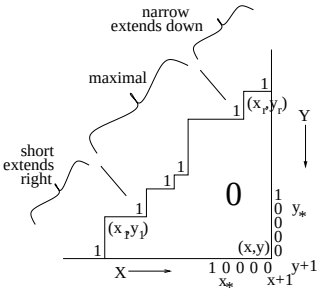

Fig. 5. Computing $staircase(x, y)$.

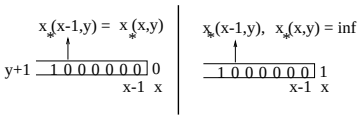

Fig. 6. Computing $x_*(x, y)$ from $x_*(x–1, y)$.

Proof. Most of the algorithm is clearly constant time per $\langle x, y\rangle$ iteration of the main loop. Two tasks however might take more time: (1) computing x_* and y_* and (2) popping the steps off the stack to check if they are maximal. We consider each task in turn.

Computing x_* and $y_* = y_r$ For each entry $\langle x, y\rangle$ in the matrix (that has a 0-entry), we must construct $staircase(x, y)$ and then determine which steps in it are maximal 0-rectangles. To construct $staircase(x, y)$, we used the values y_r and to determine which steps are maximal, we used the values x_* and y_*. To distinguish between these for different $\langle x, y\rangle$, let us now refer to these values as $y_r(x, y)$, $x_*(x, y)$, and $y_*(x, y)$. There are relationships between these values. For example, we just saw that $y_r(x+1, y) = y_*(x, y)$.

We want the time complexity of the entire algorithm to be $\mathcal{O}(nm)$ (where $n = |X|$ and $m = |Y|$). Hence, we must be able to compute each $x_*(x, y)$, and $y_*(x, y)$ in constant time. Computing them directly can take too much time. However, $x_*(x, y)$ can be computed in constant time from $x_*(x–1, y)$ and $y_*(x, y)$ can be computed similarly from $y_*(x, y–1)$. These values $x_*(x–1, y)$ and $y_*(x, y–1)$ were computed in earlier iterations and must be saved at that point in time. All previous x_* and y_* values, however, do not need to be saved, only the x_* from the previous iteration and the y_* from each $\langle x', y-1\rangle$ in the previous row of iterations.

To see how $x_*(x, y)$ can be computed from $x_*(x–1, y)$, see Figure 6. The value of $x_*(x, y)$ is the X-coordinate of the left most 0-entry in the block of 0-entries in row $y+1$ starting at entry $\langle x, y+1\rangle$ and extending to the left. So, $x_*(x-1, y)$ is the same except it considers the block extending to the left of $\langle x-1, y+1\rangle$. Therefore, if entry $\langle x, y+1\rangle$ contains a 0, then $x_*(x, y) = x_*(x-1, y)$. On the other hand, if entry $\langle x, y+1\rangle$ contains a 1, then $x_*(x, y)$ is not well defined. We could set it to $x+1$ or to ∞. Computing $y_*(x, y)$ from $y_*(x, y-1)$ is similar.

Popping steps off the stack The second task that might take more time is popping the steps off the stack to check if they are maximal, deleting them if they are too skinny, and outputting and deleting them if they are maximal. For a particular $\langle x, y\rangle$ iteration, an arbitrary number of steps may be popped. This takes more than a constant amount of time for this iteration. However, when amortized over all iterations, at most one step is popped per iteration.

Consider the life of a particular step. During some iteration it is created and pushed onto the stack. Later it is popped off. Each $\langle x, y \rangle$ iteration creates at most one new step and then only if the $\langle x, y \rangle$ entry is 0. Hence, the total number of steps created is at most the number of 0-entries in the matrix. As well, because each of these steps is popped at most once in its life and output as a maximal 0-rectangle at most once, we can conclude that the total number of times a step is popped and the total number of maximal 0-rectangles are both at most the number of 0-entries in the matrix.

It follows that the entire computation requires only $\mathcal{O}(nm)$ time (where $n = |X|$ and $m = |Y|$).

Theorem 3. *The algorithm requires $\mathcal{O}(\min(n, m))$ space.*

Proof. If the matrix is too large to fit into main memory, the algorithm is such that one pass through the matrix is sufficient. Other than the current $\langle x, y \rangle$-entry of the matrix, only $\mathcal{O}(\min(n, m))$ additional memory is required. The stack for $staircase(x, y)$ contains neither more steps than the number of rows nor more than the number of columns. Hence, $|staircase(x, y)| = \mathcal{O}(\min(n, m))$. The previous value for x_* requires $\mathcal{O}(1)$ space. The previous row of y_* values requires $\mathcal{O}(n)$ space, but the matrix can be transposed so that there are fewer rows than columns.

3.4 Number and Distribution of Maximal 0-Rectangles

Theorem 4. *The number of maximal 0-rectangles is at most $\mathcal{O}(nm)$.*

Proof. Follows directly from the proof of Theorem 2.

We now demonstrate two very different matrices that have $\mathcal{O}(nm)$ maximal 0-rectangles. See Figure 7. The first matrix simply has $\mathcal{O}(nm)$ 0-entries each of which is surrounded on all sides by 1-entries. Each such 0-entry is in itself a maximal 0-rectangle.

For the second construction, consider the n by n matrix with two diagonals of 1-entries. One from the middle of the left side to the middle of the bottom. The other from the middle of the top to the middle of the right side. The remaining entries are 0. Choose any of the $\frac{n}{2}$ 0-entries along the bottom 1-diagonal and any of the $\frac{n}{2}$ 0-entries along the top 1-diagonal. The rectangle with these two 0-entries as corners is a maximal 0-rectangle. There are $\mathcal{O}(n^2)$ of these. Attaching $\frac{m}{n}$ of these matrices in a row will give you an n by m matrix with $\frac{m}{n}\mathcal{O}(n^2) = \mathcal{O}(nm)$ maximal 0-entries.

Actual data generally has structure to it and hence contains large 0-rectangles. We found this to be true in all our experiments (see Section 4). However, a randomly chosen matrix does not contain large 0-rectangles.

Theorem 5. *Let M be a $n \times n$ matrix where each entry is chosen to be 1 independently at random with probability $\alpha_1 = \frac{N_1}{n^2}$. The probability of it having a 0-rectangle of size s is at most $p = (1 - \alpha_1)^s n^3$ and the expected number of disjoint 0-rectangle of size s is at least $E = (1 - \alpha_1)^s n^2/s$.*

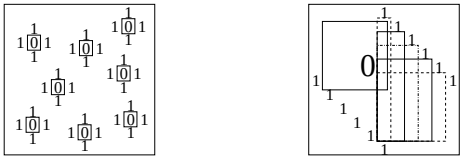

Fig. 7. Two matrices with $\mathcal{O}(nm)$ maximal 0-rectangles

Proof. A fixed rectangle of size s obtains all 0's with probability $(1-\alpha_1)^s$. There are at most n^3 different rectangles of size s in an $n \times n$ matrix. Hence, the probability that at least one of them is all 0's is at most $p = (1-\alpha_1)^s n^3$. The number of disjoint square rectangles within a $n \times n$ matrix is n^2/s. The expected number of these that are all 0's is $E = (1-\alpha_1)^s n^2/s$.

Example 1. If the density of 1's is only $\alpha_1 = \frac{1}{1,000}$ then the probability of having a 0-rectangle of size $s = \frac{1}{\alpha_1}[3\ln(n) + \ln(\frac{1}{p})] = 3,000\ln(n) + 7,000$ is at most $p = \frac{1}{1,000}$ and the expected number of 0-rectangle of size $s = \frac{1}{\alpha_1}[2\ln(n) - \ln\ln(n) - \ln(\frac{2E}{\alpha_1})] = 2,000\ln(n) - 1000\ln\ln(n) - 14,500$ is at least 1000.

As a second example, the probability of having a 0-rectangle of size $s = q \cdot n^2 = \frac{1}{1000}n^2$ is at most $p = \frac{1}{1,000}$ when the number of 1's is at least $N_1 = \alpha_1 n^2 = \frac{1}{q}[3\ln(n) + \ln(\frac{1}{p})] = 3,000\ln(n) + 7,000$. The expected number of this size is at least $E = 100$ when the number of 1's is at most $N_1 = \alpha_1 n^2 = \frac{1}{q}\ln(\frac{1}{qE}) = 2,300$.

The expected number of rectangles can be derived as a consequence of Theorem 5 as $E(s) \leq \mathcal{O}(\min(N_1 \log N_1, N_0))$ (where N_1 is the number of 1-entries and N_0 the number of 0-entries). This value increases almost linearly with N_1 as $N_1 \log N_1$ until $N_1 \log N_1 = N_0 = n^2/2$ and then decreases linearly with $n^2 - N_1$.

3.5 Multi-Dimensional Matrices

The algorithm that finds all maximal 0-rectangles in a given two dimensional matrix can be extended to find all maximal d-dimensional 0-rectangles within a given d-dimensional matrix. In the 2-dimensional case, we looped through the entries $\langle x, y\rangle$ of the matrix, maintaining the *maximal staircase*, $staircase(x, y)$ (see Figure 2). This consists of a set of *steps*. Each such step is a 0-rectangle $\langle x_i, x, y_i, y\rangle$ that cannot be extended by decreasing the x_i or the y_i coordinates. There are at most $\mathcal{O}(n)$ such "stairs", because their lower points $\langle x_i, y_i\rangle$ lie along a 1-dimensional diagonal. In the 3-dimensional case, such a maximal staircase, $staircase(x, y, z)$ looks like one quadrant of a pyramid. Assuming (for notational simplicity) that every dimension has size n, then there are at most $\mathcal{O}(n^2)$ stairs, because their lower points $\langle x_i, y_i, z_i\rangle$ lie along a 2-dimensional diagonal. In general, $staircase(x_1, x_2, \ldots x_d)$ consists of the set of at most $\mathcal{O}(n^{d-1})$ rectangles (steps) that have $\langle x_1, x_2, \ldots x_d\rangle$ as the upper corner and that cannot be extended by decreasing any coordinate.

In the 2-dimensional case, we construct $staircase(x, y)$ from $staircase(x-1, y)$ by imposing what amounts to a 1-dimensional staircase on to its side (see Figure 4). This 1-dimensional staircase consists of a single step rooted at $\langle x, y\rangle$ and extending in the y dimension to y_r. The staircase was constructed from the 1-dimensional staircase rooted at $\langle x, y-1\rangle$ by extending it with the 0-dimensional staircase consisting only of the single entry $\langle x, y\rangle$. The 1-dimensional staircase rooted at $\langle x, y-1\rangle$ had been constructed earlier in the algorithm and had been saved in memory. The algorithm saves a line of n such 1-dimensional staircases.

In the 3-dimensional case, the algorithm saves the one 3-dimensional staircase $staircase(x-1, y, z)$, a line of n 2-dimensional staircases, and a plane of n^2 1-dimensional staircases, and has access to a cube of n^3 0-dimensional staircases consisting of the entries of the matrix. Each iteration, it constructs the 3-dimensional $staircase(x, y, z)$ from the previously saved 3-dimensional $staircase(x-1, y, z)$ by imposing a 2-dimensional staircase on to its side. This 2-dimensional staircase is rooted at $\langle x, y, z\rangle$ and extends in the y, z plain. It is constructed from the previously saved 2-dimensional staircase rooted at $\langle x, y-1, z\rangle$ by imposing a 1-dimensional staircase on to its side. This 1-dimensional staircase is rooted at $\langle x, y, z\rangle$ and extends in the z dimension. It is constructed from the previously saved 1-dimensional rooted at $\langle x, y, z-1\rangle$ by imposing a 0-dimensional staircase. This 0-dimensional staircase consists of the single entry $\langle x, y, z\rangle$. This pattern is extended for the d-dimensional case.

The running time, $\mathcal{O}(N_0\ d\ n^{d-2})$, is dominated by the time to impose the $d-1$-dimensional staircase onto the side of the d-dimensional one. With the right data structure, this can be done in time proportional to the size of the $d-1$-dimensional staircase, which as stated is $\mathcal{O}(d\ n^{d-2})$. Doing this for every 0-entry $\langle x, y, z\rangle$ requires a total of $\mathcal{O}(N_0\ d\ n^{d-2})$ time.

When constructing $staircase(x, y, z)$ from $staircase(x-1, y, z)$ some new stairs are added and some are deleted. The deleted ones are potential maximal rectangles. Because they are steps, we know that they cannot be extended by decreasing any of the dimensions. The reason that they are being deleted is because they cannot be extended by increasing the x dimension. What remains to be determined is whether or not they can be extended by increasing either the y or the z dimension. In the 2-dimensional case, there is only additional dimension to check and this is done easily by reading one row ahead of the current entry $\langle x, y\rangle$. In the 3-dimensional case this is harder. One possible solution is to read one y, z plane ahead. An easier solution is as follows.

The algorithm has three phases. The first phase proceeds as described above storing all large 0-rectangles that cannot be extended by decreasing any of the dimensions (or by increasing the x dimension). The second phase turns the matrix upside down and does the same. This produces all large 0-rectangles that cannot be extended by increasing any of the dimensions (or by decreasing the x dimension). The third phase finds the intersection of these two sets by sorting them and merging them together. These rectangles are maximal because they cannot be extended by decreasing or by increasing any of the dimensions. This

algorithm makes only two passes through the matrix and uses only $\mathcal{O}(d\ n^{d-1})$ space.

Theorem 6. *The time complexity of the algorithm is* $\mathcal{O}(N_0\ d\ n^{d-2}) \leq \mathcal{O}(d\ n^{2d-2})$ *and space complexity* $\mathcal{O}(d\ n^{d-1})$.

Theorem 7. *The number of maximal 0-hyper-rectangles in a d-dimensional matrix is* $\Theta(n^{2d-2})$.

Proof. The upper bound on the number of maximal rectangles is given by the running time of the algorithm that produces them all. The lower bound is proved by constructing a matrix that has $\Omega(n^{2d-2})$ such rectangles. The construction is very similar to that for the $d = 2$ case given in Figure 7, except that the lower and the upper diagonals each consist of a $d-1$-dimensional plain of n^{d-1} points. Taking most combinations of one point from the bottom plane and one from the top produces $\Omega(n^{d-1} \times n^{d-1}) = \Omega(n^{2d-2})$ maximal rectangles.

The number of such maximal hyper-rectangles and hence the time to produce them increases exponentially with d. For $d = 2$ dimensions, this is $\Theta(n^2)$ as seen before. For $d = 3$ dimensions, it is already $\Theta(n^4)$, which is not likely practical in general for large data sets.

4 Performance of Mining Algorithm

In this section, we present two sets of experiments using our mining algorithm. The first set of experiments was designed to verify the claims of the algorithm's scalability and usefulness on large data sets. These tests were run against synthetic data. In the second set of experiments, we used a real database with data from a health insurance company. The goal of these experiments was to characterize real data sets in terms of the number, sizes, and overlaps of the discovered empty rectangles (we would expect real data sets to exhibit different characteristics than synthetic data sets such as the TPC-D benchmark). The experiments reported here were run on an (admittedly slow) multi-user 42MHz IBM RISC System/6000 machine with 256 MB RAM.

4.1 Scaling Characteristics

The performance of the algorithm depends on the number of tuples $T = |D|$ (which in matrix representation is the number of 1-entries), the number n of distinct values of X, the number m of distinct values of Y, and the number of maximal empty rectangles R. We report the runtime and the number of maximal empty rectangles R (where applicable).

Effect of T, the number of tuples, on Runtime To test the scalability of the algorithm with respect to the data set size, we held the data density (that is, the ratio of T to $n * m$) constant at one fifth. We also held n constant at 1000 since our algorithm maintains a data structure of $\mathcal{O}(n)$ size. We found these numbers to be in the middle of the ranges of the values for our real data sets. The data set is a randomly generated set of points. Initially, m is set to 500 and T to 100,000 tuples. Figure 8 plots the runtime of the algorithm with respect to the number of tuples T. The performance scales linearly as expected. The number of maximal empty rectangles also scales almost linearly with T as our analytic results of Section 3.4 predicts.

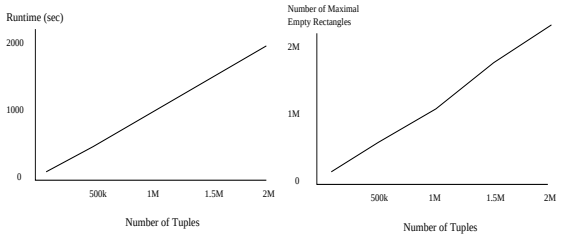

Fig. 8. Random data under constant n as T increases.

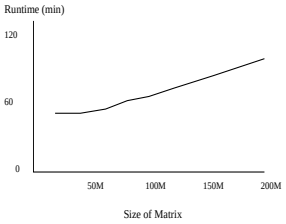

Fig. 9. Random data as the matrix size is increased (constant T).

Effect of Data Density on Runtime Note that the algorithm requires only a single pass over the data which is why we expected this linear scale up for the previous experiment. However, the in memory processing time is $\mathcal{O}(nm)$ which may, for sparse data be significantly more than the size of the data. Hence, we verified experimentally that the algorithm's performance is dominated by the single pass over the data, not by the $\mathcal{O}(n)$ in memory computations required for each row. In this experiment, we kept both T and n constant and increased m. As we do, we increase the sparsity of the matrix. We expect the runtime performance to increase but the degree of this increase quantifies the extent to which the processing time dominates the I/O. Figure 9 plots the runtime of the algorithm with respect to the size of the matrix.

Effect of R, the number of maximal empty rectangles, on Runtime Since the data was generated randomly in the first experiment, we could not precisely control the number of empty rectangles. In this next experiment, this number was tightly controlled. We generated a sequence of datasets shown for clarity in matrix representation in Figure 10.

Let $m = 1000,\ n = 2000,\ T = 1,000,000$. We start with a matrix that has 1000 colums filled with 1-entries separated by 1000 columns filled with 0-entries (for a total of 2000 columns). For each new test, we cluster the columns so that

the number of spaces separating them decreases. We do this until there is one big square 1000x1000 filled with 1-entries, and another large square 1000x1000 filled with 0s. Thus, the number of empty rectangles decreases from 1000 to 1. We would expect that R should not affect the performance of the algorithm and this is verified by the results (Figure 10).

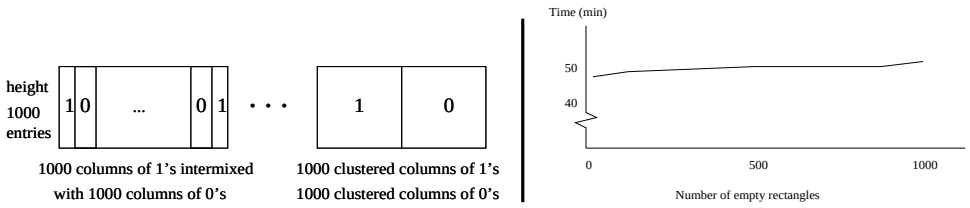

Fig. 10. Data sets and performance as the number of maximal empty rectangles R is increased.

Effect of n on the Runtime We also tested the performance of the algorithm with respect to the number of distinct X values, n. Here, we kept T constant at 1,000,000 and R constant at 1,000 and varied both n and m. To achieve this, we constructed a sequence of data sets shown again in matrix representation in Figure 11(a).

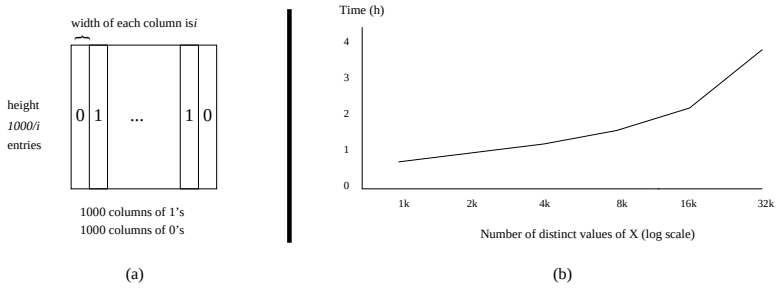

Fig. 11. Synthetic data under constant R and T. Runtime is plotted versus n.

For the first matrix, i is set to 1 so the data contains 1000 columns of 1-entries, each a single entry wide. Each column was separated by a single column of all 0's (all columns are initially 1000 entries high). In the second matrix, the height of all columns is reduced by half (to 500 entries). The width of each column (both the columns containing 1's and the columns containing 0's) is doubled. This keeps the number of tuples constant, while increasing to 2000 the number of distinct X values. The number of columns with 0's does not change (only their width increases), hence the number of discovered empty rectangles remains constant.

The process of reducing the height of the columns and multiplying their number is continued until all columns are of height 4.

The performance of the algorithm, as shown in Figure 11(b), deteriorates as expected with increasing n. Specifically, when the size of the data structures used grows beyond the size of memory, the data structures must be swapped in and out of memory. To avoid this, for data sets over two very large domains (recall that n is the minimum number of distinct values in the two attributes), the data values will need to be grouped using, for example, a standard binning or histogram technique to reduce the number of distinct values of the smallest attribute domain.

4.2 Empty Space Characteristics of Real Datasets

One of the primary motivations for our work was the observation that knowledge of empty regions can be used to great advantage in query optimization [5]. Specifically, range queries can be rewritten into more restrictive queries that can be evaluated faster. Furthermore, empty rectangles (or hyper-rectangles) can be modeled as simple materialized views in SQL. By storing such views (which amounts to storing their definition since their extents are empty), we can exploit existing work on answering queries using views to achieve these query optimization benefits. So we can rewrite queries to use empty space knowledge without changing the underlying query optimizer.

Test	n	m	T	R	Runtime (sec)	% size of largest 5 empty rectangles				
						1	2	3	4	5
1	525	8	3683	269	8	74	73	69	7	7
2	6	37716	39572	29323	78	68	58	40	37	28
3	3	1503	3061	650	6	97	94	80	12.2	0.04
4	525	423	42854	13850	86	91.6	91.6	91.3	91.3	83.1
5	14733	23292	181249	801427	10956	95	94	90	90	89

Table 1. Real data characteristics

To begin understanding the performance implications of such an approach, we sought to characterize the size and distribution of empty rectangles that occur in real data sets. We took a large relational database of health insurance data and examined the query workload. We selected queries with range restrictions on two attributes (from different relations). We then mined the underlying data for empty rectangles. The results are presented in Table 1. For each pair of attributes, we report n, m, the number of tuples in the dataset T, the total number of empty rectangles R, the time to discover all of them (Runtime), and the sizes of the five largest rectangles (the size is reported relative to the size of the dataset). In all the tests, extremely large maximal rectangles were detected. Furthermore, the query workload contained queries that ranged over large parts

of this empty space so our optimization techniques were effective [6]. For Test 1, we plot the distribution of all rectangles with respect to their sizes in Figure 12.

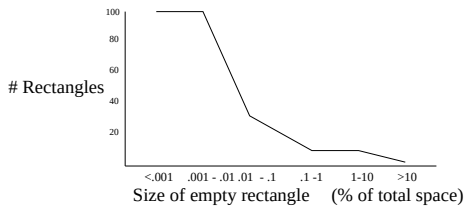

Fig. 12. Distribution of maximal empty rectangles in real data set.

5 Conclusions

We developed an efficient and scalable algorithm that discovers all maximal empty rectangles with a single scan over a sorted two-dimensional data set. Previously proposed algorithms were not practical for large datasets since they did not scale (they required space proportional to the dataset size). We presented an extension of our algorithm to multi-dimensional data and we presented new results on the time and space complexity of these problems. Our mining algorithm can be used both to characterize the empty space within the data and to characterize any homogeneous regions in the data, including the data itself. By interchanging the roles of 0's and 1's in the algorithm, we can find the set of all maximal rectangles that are completely full (that is, they contain no empty space) and that cannot be extended without incorporating some empty space. Knowledge of empty rectangles may be valuable in and of itself as it may reveal unknown correlations or dependencies between data values and we have begun to study how it may be fully exploited in query optimization [6].

References

1. R. Agrawal, T. Imielinksi, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *ACM SIGMOD*, 22(2), June 1993.
2. M. J. Atallah and Fredrickson G. N. A note on finding a maximum empty rectangle. *Discrete Applied Mathematics*, (13):87–91, 1986.
3. D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, T. Johnson, R. T. Ng, V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey Data Reduction Report. *Data Engineering Bulletin*, 20(4):3–45, 1997.
4. Bernard Chazelle, Robert L. (Scot) Drysdale III, and D. T. Lee. Computing the largest empty rectangle. *SIAM J. Comput.*, 15(1):550–555, 1986.
5. Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *Proceedings of the 25th VLDB*, pages 687–698, Edinburgh, Scotland, 1999.

6. J. Edmonds, J. Gryz, D. Liang, and R. J. Miller. Mining for Empty Rectangles in Large Data Sets *(Extended Version)*. Technical Report CSRG-410, Department of Computer Science, University of Toronto, 2000.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., New York, 1979.
8. H. V. Jagadish, J. Madar, and R. T. Ng. Semantic Compression and Pattern Extraction with Fascicles. In *Proc. of VLDB*, pages 186–197, 1999.
9. B. Liu, K. Wang, L.-F. Mun, and X.-Z. Qi. Using Decision Tree Induction for Discovering Holes in Data. In *5th Pacific Rim International Conference on Artificial Intelligence*, pages 182–193, 1998.
10. Bing Liu, Liang-Ping Ku, and Wynne Hsu. Discovering interesting holes in data. In *Proceedings of IJCAI*, pages 930–935, Nagoya, Japan, 1997. Morgan Kaufmann.
11. R. J. Miller and Y. Yang. Association Rules over Interval Data. *ACM SIGMOD*, 26(2):452–461, May 1997.
12. A. Namaad, W. L. Hsu, and D. T. Lee. On the maximum empty rectangle problem. *Applied Discrete Mathematics*, (8):267–277, 1984.
13. M. Orlowski. A New Algorithm for the Largest Empty Rectangle Problem. *Algorithmica*, 5(1):65–73, 1990.
14. T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *ACM SIGMOD*, 25(2), June 1996.