Using Approximations to Scale Exploratory Data Analysis in Datacubes

Daniel Barbará   *Xintao Wu*
George Mason University [*]
Information and Software Engineering Department
Fairfax, VA 22303
dbarbara,xwu@gmu.edu

March 4, 1999

Paper 188

Abstract

Exploratory Data Analysis is a widely used technique to determine which factors have the most influence on data values in a multi-way table, or which cells in the table can be considered anomalous with respect to the other cells. In particular, median polish is a simple, yet robust method to perform Exploratory Data Analysis. Median polish is resistant to holes in the table (cells that have no values), but it may require a lot of iterations through the data. This factor makes it difficult to apply median polish to large multidimensional tables, since the I/O requirements may be prohibitive. This paper describes a technique that uses median polish over an approximation of a datacube, easing the burden of I/O. The results obtained are tested for quality, using a variety of measures. The technique scales to large datacubes and proves to give a good approximation of the results that would have been obtained by median polish in the original data.

1 Introduction

Exploratory Data Analysis (EDA) is a technique [13, 14, 22] that uncovers structure in data. EDA is performed without any a-priori hypothesis in mind: rather it searches for "exceptions" of the data values relative to what those values would have been if anticipated by an statistical model. This statistical model fits the rest of the data values rather well and can be accepted as a description of the dataset. When applied to multidimensional tables, EDA also uncovers the attribute values that have the greatest effect on the data points. As pointed out by [19],

[*]This work has been supported by NSF grant IIS-9732113

these exceptions can be used by an analyst as starting points in the search for anomalies, guiding the analyst work across a search space that can be a very large.

A widely used data model for On-Line Analytical Processing (OLAP) is the datacube [12]. A datacube is a multidimensional data abstraction, where aggregated *measures* of the combinations of dimension values are kept. For instance, in a company datacube, the dimensions can be product names, store names and dates. The measure can be the total sales in dollars. A cell of the datacube corresponds to a combination of the dimension values and contains an aggregate of the measure: e.g., the cell of product #001, store 'AAA', during the month of May, 1998, contains the total amount of sales of that product, in that store for that month. Datacubes define cells at all levels of aggregation, e.g., sales for **ALL** the products in store 'AAA' during May, 1998. Moreover, dimensions have associated hierarchies that specify other aggregation levels, e.g., stores can be grouped individually, by city, by state, etc.

At any level of aggregation, datacubes can be viewed as *multi-way tables*, that can be subjected to EDA. Two traditional ways of performing EDA in tables are *median polish* and *mean polish* [14]. Both methods try to fit a model (additive or multiplicative) by operating on the data table, finding and subtracting medians (means) along each dimension of the table. Starting with one dimension, the method calculates the median (mean) of each "row" [1] and subtracts this value from every observation in the row. Then, the method moves to the next dimension and uses the table resulting from the previous operation to find medians (means) in each row and subtract them from the entries in this row. Of course, doing that changes the medians (means) on the previous dimension rows. The process continues with each dimension and iteratively cycles through the set of dimensions. In principle, the process continues until all the rows in each dimension have zero median (mean), hence the number median (mean) polish. (In practice, the process stops when it gets sufficiently close to that goal.) One gets two kinds of information from this process. First, one gets the *effect* that each row in each dimension has on the model, given by the algebraic sum of the medians (means) that have been subtracted in that row at every step. Secondly, one gets *residuals* in each cell of the table, which tell us how far apart that particular cell is from the value that would have been predicted by the model being fit.

It has been pointed out that the main drawback of the mean polish method is its lack of resistance to outliers (i.e., cells that do not fit the model well). This lack of resistance manifests itself specially when "holes," e.g., missing cells, are present [14]. This is particularly troublesome for datacubes, which are usually sparse (i.e., not every cell has a value). On the other hand, median polish, being more resistant to outliers and holes, is very adversely affected by holes which increase substantially the number of iterations needed by the process [14]. Increasing the number of iterations can drastically impose enormous I/O demands (by requesting many passes over a large dataset) and therefore render the process impractical

[1] we use the term "row" to refer to the set of cells in the hypercube which share the same attribute value for one of the dimensions

	Race		
Age	*White*	*African American*	*Hispanic*
10-14 years	0.8	4.2	2.7
15-17 years	30.0	69.7	72.9
18-19 years	81.2	137.1	157.9

Figure 1: Birth rates to teenagers, by age and race: United States, 1995.

for large datacubes. Previous work in using EDA for datacubes [19] has chosen to use mean polish, precisely for being less demanding on the number of iterations needed to finish.

This paper explores a method that benefits from the high robustness of median polish, while at the same time scales well for large datacubes. Our method uses statistical models to approximate subsets of the datacube, eliminating the need to do several passes over the dataset in order to compute the medians. However, by doing so, the row effects and the residuals computed are only an approximation of those that would have been computed by applying median polish to the dataset. We show that this tradeoff of lesser I/O demands versus accuracy in the results is very beneficial. The anomalies pointed out by our approximate method are very close to those pointed out by applying the method to the real dataset, while the resulting process scales well to large datacubes.

This paper is organized as follows. In Section 2, we present the median polish method and illustrate its results via a simple example. In Section 3, we present the basis of our methods, including the algorithms and data structures utilized in them. Section 4 presents the experimental results obtained by our method using several real and synthetic datasets. Section 5 summarizes the related work. Finally, Section 6 presents the conclusions and directions of future work.

2 Median Polish Revisited

In this section we briefly review the median polish method and present an example, in order to help clarify the concepts presented in the rest of the paper.

To illustrate the procedure, we use a simple example taken from from [16]. Figure 1 shows a simple two-way table of birth rates to teenagers in the year 1995, by three age groups and race and Hispanic origin [2].

The relationship between the rate (response variable) and the two factors in the table (race and age) can be expressed using many models. A simple, *additive model* can be written as shown in Equation 1, where μ is the overall typical value for the whole table (common value), α_i is the *row effect* of row i, β_j is the *column effect* of column j and ϵ_{ij} is the departure of y_{ij}

[2]Hispanic is not really a race: people of Hispanic origin may be of any race.

	Race				
Age	*White*	*African American*	*Hispanic*	new median	effect
10-14 years	36.3	0.0	-4.7	(1.5)	-67.0
15-17 years	0.0	0.0	0.0	(0.0)	0.0
18-19 years	-16.2	0.0	17.6	(0.0)	67.4
new median	(0.0)	(0.0)	(0.0)	(0.0)	(0.0)
effect	-39.7	0.0	0.0	0.0	69.7

Figure 2: Final table.

from the model or *residual*.

$$r_{ij} = \mu + \alpha_i + \beta_j + \epsilon_{ij} \tag{1}$$

Figure 2 shows the table after the median polish procedure. (We omit the intermediate steps for lack of space.) All row and column medians have reached a value of 0. The "previous" column and row contain the effects of the rows and columns respectively. The cell on the lowest, right-hand corner contains the total effect. So, we can see that teenager birth rates are lowest among whites (column effect $\beta_1 = -39.7$), lowest among the 10-14 age range (row effect $\alpha_1 = -67$), and highest among the 18-19 age range (row effect $\alpha_3 = 67.4$). We also see that the cell that least fits the model is that of White teenagers on the age range 10-14 (residual $\epsilon_{11} = 36.3$). These parameters explain the influence of each factor with respect to the overall data, allowing the analyst to make meaningful conclusions (e.g., teenage birth rates are lowest among whites, when compared with the other race groups).

It is easy to see that this method cannot scale well. During each iteration, we need full access to the entire dataset. If the dataset does not fit in main memory, this implies severe I/O demands: a row of the table would be brought to memory only to be replaced by other portions of the dataset later, and brought back in the next iteration. Our method aims to eliminate the need for all these I/O at the expense of working with data approximations.

3 Our method

In this section we describe in detail how we scale the median polish procedure to large datacubes. The aim of our method is to avoid having to bring data cells to main memory repeatedly in order to compute the medians. We do that by characterizing portions of the core cuboid of the datacube [3], employing statistical models that can be later used to estimate

[3]the core cuboid is the set of cells of finest granularity; any other cuboid in the datacube can be computed from the core cuboid by aggregating on one or more of the dimensions.

the values of the individual cells. To avoid incurring in large errors by using the estimated cell values, we retain all the cell values whose estimated values are too erroneous (farther away from the real value by more than a preestablished threshold). We then keep the model parameters (for each portion of the core cuboid) in main memory, along with the retained set of cells (or at least as much of these values as we can fit in memory) and start the median polish procedure. In each iteration, the procedure will use, for a given cell one of the following values: a) the estimated value given by the corresponding model, if the cell is not retained or, b) the actual cell value, if the cell has been retained. Hence, this method minimizes I/O by not having to fetch cells from the disk too often (if all the retained cells fit in main memory, the I/O requirements after modeling the portions of the core cuboid drop to zero). Of course, the usage of the estimated value brings as a consequence that the results of the median polish procedure are only approximate. However, as we shall show in the results section, the results are extremely close to performing median polish in the real datacube, even when large estimation errors are allowed.

It is important to point out that while our descriptions and experiments have been performed using the core cuboid, it is straightforward to use our data structures and method to perform approximate EDA in any other cuboid of the datacube. The models used to describe portions of the core cuboid, can be used to perform aggregations to obtain cells in any other cuboid, and our method can then operate in these aggregations.

The issues involved in performing our approximate method can be summarized as follows:

- Selecting chunks of the core cuboid that will be described by models, selecting the model to use to characterize them, and deciding which cells need to be retained.
- Organizing the model parameters and retained cells to efficiently access them during the median polish run.
- Evaluating the quality of the results obtained by the approximate method.

In the rest of this section, we describe in detail how we solved each of these issues.

3.1 Dividing the core cuboid

In order to select the chunks in which we divide the core cuboid we use a density based approach called hierarchical clustering [15] to get high density portions of the cube. This approach has been previously utilized to identify regions of high density in multidimensional data ([1]). (The aim in [1] is to have a density approach to clustering, where a cluster is defined as a region with higher density of points than its surrounding regions.) We assume that the core cuboid has been computed (an algorithm such as the one presented in [18] is well suited for the task).

Given a $d-$dimensional datacube with dimensions $A = \{A_1, A_2, \cdots A_d\}$, we assume it be a set of bounded, totally ordered domains space[4]. Accordingly, $S = \|A_1\| \times \|A_2\| \times \cdots \times \|A_d\|$, is the possible number of cells in the core cuboid. The non-empty cells in the cuboid are a set of multidimensional points, $V = \{\hat{v_1}, \hat{v_2}, \cdots, \hat{v_m}\}$. where each $\hat{v_i} = \{v_{i_1}, v_{i_2}, \cdots, v_{i_d}, m_i\}$, with v_{i_j} the i-th dimension value and m_i the value of the cell.

Initially, we partition the space of the cuboid into non-overlapping rectangular 1st-level chunks which have the same chunk size $\{\mu_1^1, \cdots, \mu_d^1\}$. Hence, the number of chunks in 1st-level is $C_{no}^1 = \prod \lceil \frac{\|A_i\|}{\|\mu_i^1\|} \rceil$. We use c_i^1 to denote the i-th chunk in the first level. The size of any of the 1st-level chunks is given by $d_i = \prod \mu_i^1$. Clearly, we require that $\mu_i^1 \leq A_i$ for all i; moreover, we choose not to divide those dimensions with small domains (for them we make $\mu_i^1 = A_i$). At any point during the process of partitioning we may decide to further divide a 1-st level chunk into several 2nd level chunks, and successively an j-th level chunk into $j+1$-th level chunks. The size of an j-th level chunk is predetermined to be $\{\mu_1^j, \cdots, \mu_d^j\}$, for $j = 1, \cdots, MAXLEVEL$, where $MAXLEVEL$ is the maximum level of any chunk.

Before we formally present the algorithm for cuboid partitioning, let us describe the data structures that are needed to make it work. A chunk is described as a structure which contains the following fields (each with the obvious meaning): *Chunk-number, Number-cells, Number-outliers, Level, State, Pointer-to-parent, Parameter-list, Cell-list, Outlier-list.*

In particular, *State* defines the state of the chunk, which changes dynamically as the partitioning process runs. The possible states where a chunk can be found are as following:

- STAT-NULL : no cells are located in this chunk. This is the initial state of every chunk and the final state for empty chunks.
- STAT-SPARSE : a chunk with very few cells in it. (We just retain all cells in this chunk.)
- STAT-DENSE : a chunk with enough cells which can be modeled.
- STAT-REDV : a chunk with intermediate density (enough cells not to be sparse, but not enough to be modeled yet), or a chunk where a model fit is not satisfactory. This are chunks subjected to further partitioning to the next level.
- STAT-MODL : this chunk has been modeled.
- STAT-DVOK: this chunk has been further subdivided.

After the partitioning procedure, every chunk should be in STAT-NULL, STAT-SPARSE or STAT-MODL. The rest are intermediate states.

[4]Without loss of generality and to keep the presentation simple, we assume in this paper that the actual attribute values are consecutive integer values from 0 to $\|A_i\| - 1$ where $\|A_i\|$ is ith-dimension attribute domain size , For the general case where the actual attribute values are not necessarily consecutive, we can map each actual attribute value to its rank.

At every step of the partitioning process we maintain a CHUNK-DESC list in memory which contains information about every chunk's level, state, and count of cells. Moreover, there are three parameters used to drive the partitioning process:

- α : this marks the minimum acceptable density for chunks measured by the number of cells in the chunk divided by the chunk's size.
- β : this is the maximum error level tolerated in the estimation process. That is if y is the value of a cell and $\hat{y}$ the value of the estimation by the model describing the chunk, the relative error given by $|y - \hat{y}|/y$ must be less than or equal to β. Any cell whose estimation error surpasses β is declared an outlier.
- γ : this is the maximum percentage of outlier cells allowed in the chunk, which is measured as the number of outliers divided by the number of cells in the chunk. If the percentage of outlier cells is bigger than γ, we declare the chunk's state as STAT-REDV and proceed to partition it further.

Figure 3 presents the pseudo-code for the partitioning algorithm. As can be noticed, the algorithm proceeds to divide the initial cuboid in 1-st level chunks, assigning cells to each one and proceeds to classify each chunk, further dividing it if necessary, until the chunk is declared STAT-SPARSE, STAT-NULL or STAT-MODL. It is worth noticing that the algorithm will read the number of cells in the core cuboid into memory only once. Then each chunk that is neither sparse not null will be read into memory (with its respective cells) and processed (subdivided, modeled) until no more processing is needed for the chunk. If each chunk fits in memory, then it is guaranteed that the each cell will be read into memory only once, making the input activity equivalent to two passes of the data and the write activity also equivalent to two passes of the data. If a chunk does not fit in memory several reads and writes to the chunk will be needed, making the total read activity loosely bound by $MAXLEVEL$ number of passes through the data. (In practice the I/O activity will be much less, since there will be child chunks that fit in memory.) Moreover, we need to point out that all this I/O activity takes place before the EDA is undertaken. (Chunks and their models can be stored as part of the cuboid and reused for further analysis and other uses, such as approximate query processing and other types of data mining [5, 6, 3].)

A point we need to make in this subsection is that of modeling. Several choices of models are possible (a survey of methods can be found in [4]). The general technique is, however, quite independent of the model chosen for the chunks. Of course, there will be some models that are better suited for specific classes of data and therefore will produce smaller estimation errors. However, as we will show in Section 4, the results of the approximate median polish are quite robust, in spite of the errors incurred by the modeling process. For our prototype, we chose a simple linear regression model in the number of dimensions. The regression was computed using the marginal values (total sum of measure values per "row" of each dimension).

```
$Partition(core - cuboid)$
    $BEGIN$
        $Divide(core - cuboid)$
        $while$ there is a chunk $C$ in the CHUNK-DESC list such that
            $C.Stat \in \{$ **STAT-DENSE, STAT-REDV** $\}$
            $switch$ $(C.Stat)$

                $case$ '**STAT-REDV**'

                    $if$ $C.level < MAXLEVEL$
                      $C.Stat =$ **STAT-DVOK**
                      $Divide(C)$
                    $else$
                      $C.Stat =$ **STAT-SPARSE**

                $case$ '**STAT-DENSE**'
                    $Model(C)$
                    $If$ $(C.Number\text{-}outliers/C.Number\text{-}cells) > \gamma$
                      $and$ $C.Level < MAXLEVELS$
                      $Divide(C)$
                    $else$
                      $C.Stat =$ **STAT-MODL**
    $END$

$Divide(CHUNKC)$
    $Begin$
        $For$ each cell in chunk $C$ do
            include cell the corresponding child chunk $C'$
            increase $C'.Number\text{-}cells$

        let $d$ be the size of any child chunk
        $for$ each child chunk $C'$ do
            add $C'$ to DESC-LIST.
            $if$ $C'.Number\text{-}cells = 0.$
                $C'.Stat =$ **STAT-NULL**.
            $else$
                $if$ $C'.Number\text{-}cells/d \geq \alpha.$
                    $C'.Stat =$ **STAT-DENSE**.
                $else$
                    $C'.Stat =$ **STAT-SPARSE**.
    $END$
```

Figure 3: Partitioning Algorithm

Special mentioning should be made of the treatment of holes for chunks whose state is *STAT-MODL*. (Holes in chunks whose state is *STAT-SPARSE* are taken care of by definition: only a list of non-zero cells is kept on those.) Since it is assumed that the number of holes of a *STAT-MODL* type of chunk is small (since α must be big), we can simply treat them as outliers and keep them in the outliers list. Otherwise, we need a separate index to indicate which cells are non-zero. This has repercussions in our median polish method: the smaller the chunk descriptions are, the more of them we will be able to keep in memory, thereby avoiding the need for fetching them from the disk.

3.2 Median Polish Algorithm Using the Chunk Models

In Figure 4 we present the algorithm to perform the median polish process by using the chunk characterization. First, both the CHUNK-DESC list and a number of chunk structures (as many as the memory buffer allocated for such purpose allows) are prefetched into main memory (lines 1 and 2). Then the effects for every row of every dimension are initialized to 0 (line 3). The process of polishing medians is conducted until a predetermined number of steps is carried out ($MAX - STEP$) or more than τ rows have median equal to 0 (line 10). (In practice, we do not demand that the median be equal to zero, but rather less than a small value.) Then the for loops of lines 4 and 5 drive the median polish computation. First, a query q is set up to describe all cells belonging to row j of dimension i ($q = (A_i = j)$) (line 6). Then the procedure $get - residual$ is called for that row (using q): $get - residual$ maps q to the chunks that contain data that satisfy q (to avoid going through every chunk). Notice that the chunks at this stage can be of state *STAT-SPARSE* or *STAT-MODL*. In case the chunk used is of the type *STAT-SPARSE*, the algorithm reads the real cell values, stored in the chunk structure, otherwise (if the chunk is of the type *STAT-MODL*, the cell values can be either estimated by using the model parameters, or read if the cell happens to be an outlier. As cells are used or estimated (i.e., taken from the outliers or modeled), they are placed in the set r (lines 19 and 24). Once all the chunks in the list have been processed, the cells in r are processed one by one, subtracting to the cell value all the median values for all the dimension attribute values that define the cell (lines 25, 26 and 27). In this way, the values kept in the chunks correspond, at every step, to those of the original cuboid. This makes it unnecessary to write a chunk to disk, if we need to reclaim the buffer space occupied by it. After subtracting the medians, the cell value reflects the residual at that step of the computation. With these values, we compute the median of the row (line 8) and subtract that from the corresponding effect (line 9). If the entire set of values r does not fit in memory, we can perform the classic divide-and-conquer approach used to compute order statistics, which divides the list in k sequences of $\|r\|/k$, computes the medians of the sequences and the median of the sequence of medians and recursively calls the procedure to find the overall median (see [2, 7]).

We want to revisit the issue of the chunk description size here. As we said in Section

```
median-polish()
BEGIN
        read CHUNK-DESC list to memory (1)
        prefetch chunks to memory (2)/* as many as can be held*/
        make $eff_{ij} = 0$ for $i \in \{1, 2, \cdots, d\}$ and $j \in \{0, 1, \cdots, \|A_i\| - 1\}$
        for (step=0; step < $MAX - STEP$;step++) (3)
            forall dimension $i \in \{1, 2, \cdots, d\}$ (4)
                for ( $j = 0; j < \|A_i\| - 1$) (5)
                    $q = (A_i = j)$ (6)/* set a query $q$ to find cells for which $A_i = j$.*/
                    $r =$ get-residual(QUERY q); (7)
                    $median_{ij} = median(r)$(8)
                    $eff_{ij} += median_{ij}$ (9)
            if $count(median = 0) > \tau$ (10)
            break (11)
        get-effects() (12)/*compute the common value and effects*/
END

get-residual(QUERY q)
BEGIN
        $r = \emptyset$ (13)/* initialize the residual set */
        get chunks C which involved in the QUERY q (14)
        forall chunk c $\in C$ (15)
              switch ($c.Stat$) (16)
                    case 'STAT-SPARSE'
                          if c is not in memory, read it from the disk (17)
                          for every cell $z$ in $c$ that satisfies $q$ (18)
                              put $z$ in $r$ (19)
                    case 'STAT-MODL'
                          if c is not in memory, read it from the disk (20)
                          forevery cell $z$ in $c.outliers$ that satisfies $q$ (21)
                             put $z$ in $r$ (22)
                          for every cell $z$ not in $c.outliers$ such that $c$ satisfies $q$ (23)
                              estimate $z.value$ and put $z$ in $r$ (24)
        for every $z \in r$ (25)
              for every dimension $i \in \{1, 2, \cdots, d\}$ (26)
                  let $z.i = j$ (27)
                  $z.value -= eff_{ij}$ (28)
END
```

Figure 4: Median Polish Algorithm

3.1, the ability to keep many (or all) the chunks in memory decreases with the size of the chunks. This implies that lines 17 and 20 of the algorithm of Figure 4 will not be executed very frequently if the chunks are small. This, of course, assumes that the regions covered by chunks whose state is *STAT-MODL* are indeed dense (i.e., contain few holes). If this is not the case, the zero (or non-zero [5]) cells need to be indexed, making the description larger. So our procedure is more effective for data that is highly clustered in subspaces along the datacube. Fortunately, this is the case in many real datasets. (Commercial systems exploit this property: for instance, Arbor Software's Essbase, stores dense and sparse regions of the core cuboid using different data structures [9, 10]). However, as the results will show, we still get a substantial benefit in performance if the density in the clusters is low.

3.3 Quality Measures

Since our procedure is approximate, we need to define ways to compare the results with those that are obtained by applying the median polish method to the base cuboid. In this subsection we describe the measures we use for that purpose.

The first measure has to do with the error incurred in the calculations of the common effect by using the models instead of directly calculating medians over the data. Equation 2 shows the relative error for the common effect.

$$CErr = \frac{\|C - \hat{C}\|}{\|C\|} \tag{2}$$

More important than the relative errors made in the computation of row effects and residuals is the rank that a particular effect (or residual) has, in an order list of effects (or residuals). After all, we want to direct the attention of the user to those effects (residuals) which are large. (In an ordered list of effects, based on their values, one is presumably very interested in those which rank very high -large positive effects- or very low -large negative effects-.) To measure the quality of our results regarding the ranks, we define the ordered sets $E_i = \{e_{i0}, e_{i1}, \cdots, e_{in}$ where $n = \|A_i\| - 1$ and e_{ij} is the rank of effect j in dimension i, as the set of dimension i ranks. Given these sets, Equation 3 defines the average rank slipperage ($AveRSl$), as the mean of the relative differences in rank between effects computed by using median polish in the original cuboid (e_{ij}) and approximate cuboid ($\hat{e_{ij}}$).

$$AveRSl_i = \frac{sum_j(\frac{\|e_{ij} - \hat{e_{ij}}\|}{e_{ij}})}{n_i}, \text{ for } \quad i = 1, 2, \cdots, d \;\; j = 0, 1, \cdots, n_i, \quad n_i = \|A_i\| - 1. \tag{3}$$

Let us define the sets E_i^+ and E_i^- as the sets of the k topmost ranked effects and the k lowest ranked effects in row i respectively, where $k = \| E_i^+ \| = \| E_i^- \|$. Given the

[5]the choice of which one we should index depends on which ones are fewer.

equivalent sets $\hat{E}_i^+$ and $\hat{E}_i^-$, obtained by the approximate median polish, we can define the positive and negative effect recall (Re^+, Re^-) and the positive and negative effect precision (Pe^+, Pe^-) in the manner shown in Equations 4,5,6, and 7.

$$Re_i^+ = \frac{\| E_i^+ \cap \hat{E}_i^+ \|}{\| E_i^+ \|}, \text{ for } i = 1, 2, \cdots, d \tag{4}$$

$$Re_i^- = \frac{\| E_i^- \cap \hat{E}_i^- \|}{\| E_i^- \|}, \text{ for } i = 1, 2, \cdots, d \tag{5}$$

$$Pe_i^+ = \frac{\| E_i^+ \|}{\| \hat{E}_i^+ \|}, \text{ for } i = 1, 2, \cdots, d. \tag{6}$$

$$Pe_i^- = \frac{\| E_i^- \|}{\| \hat{E}_i^- \|}, \text{ for } i = 1, 2, \cdots, d. \tag{7}$$

Finally for the residuals we define the set R as the set of the k-th largest residuals, with $\| R \| = k$ computed by the standard median polish and $\hat{R}$ defined for the residuals obtained with the approximate cuboid. Then we define the Recall (Rec) and Precision $(Pres)$ measures in Equations 8 and 9 respectively.

$$Rec = \frac{\| R \cap \hat{R} \|}{\| R \|}, \text{ with } \| R \| = \| \hat{R} \| \tag{8}$$

$$Pres = \frac{\| R \|}{\| \hat{R} \|}, \text{ such that } \| \hat{R} \| \text{ is the minimum value for which } R \subseteq \hat{R}. \tag{9}$$

4 Results

The experiments were conducted in a SUN Ultra 2, with two processors, and 500 Mbytes of RAM. We conducted the experiments in three datasets. The results follow.

4.1 Dataset 1

The first dataset is a real dataset from a retailer application. The set contains 4,933 tuples and three dimensions of cardinality 42, 61, and 18.

The experiment in dataset 1 was conducted with an $\alpha = 0.01$ and $\gamma = 0.2$. Figure 5 shows the values for $CErr$ and $AveRSl_i, i = 1, 2, 3$ for the first dataset. Both the relative error for the common effect and the average slipperage for the effects remain low even for $\beta = 0.4$, reflecting the very good quality of the results obtained by the approximate method.

Figure 6 shows the positive and negative effect recall and precision for dataset 1. For the sake of brevity in the table, this is done only for one β values (0.1).

β	$CErr$	$AveRSl_1$	$AveRSl_2$	$AveRSl_3$
0.1	0.090	0.194	0.364	0.340
0.2	0.107	0.270	0.528	0.399
0.3	0.108	0.317	0.601	0.471
0.4	0.108	0.343	0.629	0.476

Figure 5: Relative errors for common effect and average rank slipperage for dataset 1

Figure 7 shows the recall and precision figures for dataset 1. In each case, the experiments were done with four different values for β, to show the trends as we choose to store less outliers. Notice that the recall values remain high for the entire set of experiments. For βs of 0.1 and 0.2 they are all above 90% with the exception of the value obtained for $|||R||| = 500$ and $\beta = 0.2$. Even for models that tolerate higher errors (higher βs), the values are very high. This shows that most of the larger residual effects found in the standard median polish will be captured by an equal sized set of residuals in the approximate method. The precision results show that if $beta$ is kept at 10%, we will only need to look at a set that is at most 15% (i.e., 1/0.87 in $||R|| = 200$) larger than the set R, and in most cases this value would be significantly smaller. For higher βs and a larger set R, the precision gets progressively worse. (It is important to keep in mind that the user is likely to be interested in a few residuals -the largest ones-.)

4.2 Dataset 2

The second dataset used is taken from the U.S Census Bureau data [8]. It is a two dimensional table that contains the population for 227 countries from the year 1950 to the year 2049 (future years had been projected by the Census Bureau). There are 22700 tuples in it and no holes.

The experiment in dataset 2 was conducted with $\alpha = 0.01$ (which is really irrelevant here, since the set has no holes) and $\gamma = 0.2$. The relative error of the common effect and the average slipperage for the two dimensions of the dataset are presented in Figure 8. As in the previous dataset, the relative errors and slipperage values are small.

Figure 9 shows the positive and negative effect recall and precision values for dataset 2. Again, only one value of $\beta = 0.1$ is reported in the table.

Figure 10 shows the recall and precision values for dataset 2 (again for four different values of β). The results for recall are again very high across the spectrum of experiments, showing that most of the largest residual effects of the standard method will be captured by an equal sized set of residuals in the approximate method. Precision values are high for the the first two rows and two β values. The rest indicate that a larger residual set is needed in the approximate method to capture all the largest effects on the standard median polish.

k	Re_1^+	Re_2^+	Re_3^+	Re_1^-	Re_2^-	Re_3^-	Pe_1^+	Pe_2^+	Pe_3^+	Pe_1^-	Pe_2^-	Pe_3^-
5	0.6	0.6	0.6	0.8	0.2	0.8	0.385	0.263	0.5	0.385	0.167	0.833
10	0.6	0.5	0.9	0.7	0.5	0.7	0.556	0.526	0.91	0.589	0.244	0.667
15	0.8	0.667		0.8	0.66		0.75	0.652		0.83	0.365	
20	0.95	0.85		0.95	0.7		0.645	0.513		0.95	0.488	
25		0.80			0.72			0.610			0.595	
30		0.767			0.80			0.714			0.714	

Figure 6: Positive and negative effect recall and precision values for dataset 1 (k is the size of the set of largest positive or largest negative effects; empty values reflect the fact that the set of positive or negative effects is smaller than the corresponding k.)

	Rec				$Pres$			
$\parallel R \parallel$	$\beta = 0.1$	$\beta = 0.2$	$\beta = 0.3$	$\beta = 0.4$	$\beta = 0.1$	$\beta = 0.2$	$\beta = 0.3$	$\beta = 0.4$
50	0.96	0.98	0.94	0.94	0.93	0.91	0.89	0.88
100	0.98	0.96	0.95	0.94	0.92	0.84	0.81	0.76
150	0.95	0.93	0.92	0.92	0.79	0.76	0.71	0.70
200	0.98	0.97	0.94	0.93	0.99	0.87	0.80	0.75

Figure 7: Recall and Precision for the residuals of dataset 1 for four different β values.

4.3 Dataset 3

This is a synthetic dataset whose parameters can be changed to drive a series of experiments. It is important to point out that we use this dataset to test the scalability of the method as the number of non-zero cells in the datacube grows. The quality measures for this dataset are of secondary importance. The dataset has four dimensions with corresponding cardinalities 320,160,80 and 40. The non-zero cells values are distributed with a normal distribution of mean 20 and variance 4. We perturb the distribution by adding to the value of the cell a number which depends on the cell's position in the cube. (We do this to make the values depart from the normal distribution.) We choose the size of the dataset in number of non-zero cells, N_c, the size of a chunk s_c and the density of the dense chunks α. Those values determine the number of dense chunks, N_{dc} as $\lfloor \frac{N_c}{\alpha * s_c} \rfloor$. The remaining non-zero cells not attached to a dense chunk are randomly placed in the remaining (spare) chunks. Also, the placement of dense chunks is done randomly. We conducted all the experiments with a $\beta = 0.2$, $\gamma = 0.2$, and two values of α (0.33 and 0.95).

Figures 11 plots the execution time of the standard method and the approximate method for chunk densities of 33% and 95%. As can be observed, the approximate method scales well with the size of the dataset, while the execution time of the standard method gets to be impractical for large sizes (more than 10 hours for 2 million non-zero cells). It is also worth pointing out that in the high density case, the approximate algorithm performs better

β	$CErr$	$AveRSl_1$	$AveRSl_2$
0.1	0.032	0.024	0.040
0.2	0.033	0.024	0.041
0.3	0.025	0.027	0.027
0.4	0.108	0.031	0.027

Figure 8: Relative errors for common effect and average rank slipperage for dataset 2

k	Re_1^+	Re_2^+	Re_1^-	Re_2^-	Pe_1^+	Pe_2^+	Pe_1^-	Pe_2^-
10	1	1	1	0.60	1	0.625	1	1
20	0.95	0.95	1	0.80	0.714	0.689	1	0.952
30	1	1	0.97	0.933	1	0.937	0.938	1
40	0.975	1	1	1	0.975	1	1	1
50	1	1	1	1	1	1	1	1
60	0.983		1		0.984		1	
70	1		0.985		1		0.972	
80	1		0.988		1		0.988	
90	1		1		1		1	
100	0.99		1		0.99		1	

Figure 9: Positive and negative effect recall and precision values for dataset 2.(k is the size of the set of largest positive or largest negative effects.)

	Rec				$Pres$			
$\parallel R \parallel$	$\beta = 0.1$	$\beta = 0.2$	$\beta = 0.3$	$\beta = 0.4$	$\beta = 0.1$	$\beta = 0.2$	$\beta = 0.3$	$\beta = 0.4$
50	0.92	0.92	0.68	0.68	0.79	0.79	0.53	0.53
100	0.96	0.96	0.85	0.85	0.90	0.90	0.23	0.23
150	0.88	0.88	0.80	0.79	0.58	0.58	0.05	0.05
200	0.92	0.92	0.78	0.77	0.42	0.42	0.06	0.06

Figure 10: Recall and Precision for the residuals of dataset 2 for four different β values.

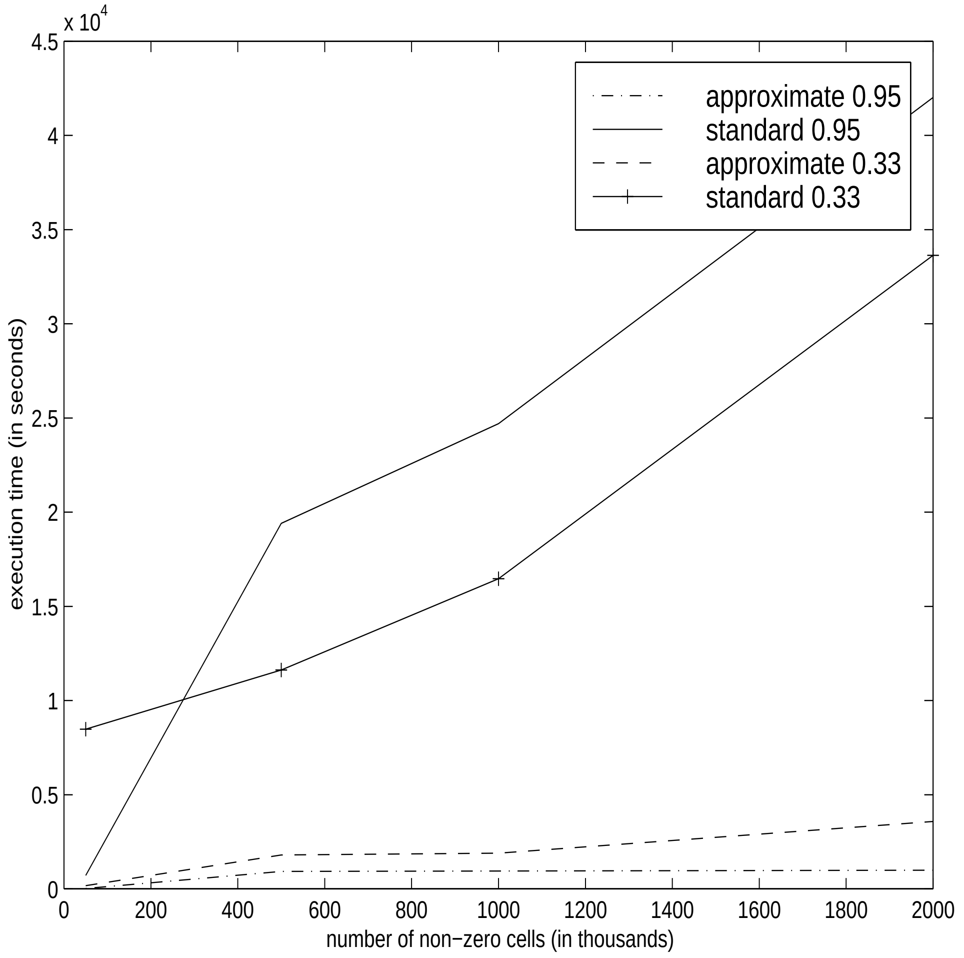

Figure 11: Execution time for the standard and approximate method for dataset 3. The Y axis shows the execution time in seconds; The X axis shows the number of non-zero cells in thousands.

since there is no need to index the non-zero cells in the chunk. Nevertheless, in the case of low density, the approximate algorithm still outperforms the standard by a ratio of 10:1 (as opposed to a ratio of more than 30:1 in the dense case).

Figure 12 shows the number of steps taken by the methods over dataset 3 for two different densities in *STAT-MODL* type chunks: 33 % and 95 %, for both the approximate method and the standard method. The table shows also the time to build the models in the case of the approximate method. We can appreciate that the model building time is less than 10 % of the time taken to do the approximate median polish for the larger datasets. Also, we see that the number of steps taken by the approximate and standard method are comparable in most cases.

Finally, Figures 13, 14 and 15 show the quality measures for one of the sizes of dataset 3. (Table 14 shows only the results for 3 dimensions, to keep the table small; the 4-th dimension results are similar to the others.) The results obtained for other sizes are very similar.

Number of non-zero cells	*density*							
	0.95				0.33			
	approximate			*standard*	*approximate*			*standard*
	N_c	t_b	steps	steps	N_c	t_b	steps	steps
50	10	4.97	3	4	27	6.97	6	46
500	100	58.14	16	10	290	86.46	8	6
1,000	202	116.04	10	6	566	215.36	4	4
2,000	404	267.27	5	5	1128	492.18	4	4

Figure 12: Number of dense chunks, steps and time to build models for each dataset size (in thousands of non-zero cells) for two different chunk densities.

5 Related Work

Using median polish for fitting additive models in a two-way table was described, along other methods to do exploratory data analysis in the first edition of [22]. More recent books like [13, 14] describe median and mean polish and show their use with real datasets. It is generally agreed that polishing with medians is more resistant to outliers and holes than the mean polishing method. Tukey [21], includes a general discussion of polishing methods. Methods of fitting that minimize a sum of absolute deviations were proposed initially by [17, 20]. Gentle [11] discusses fast computation algorithms for this approach. Mean polish provides the solution to a least-squares problem that aims to minimize the sum of squared residuals. However, when there are holes in the data, the least-squares solution does not correspond to that given by mean polish.

There is very little work addressing the scalability of polishing methods in datacubes. Sarawagi et al [19] address this issue and develop an algorithm to scale mean polish to large datasets. However, they choose to ignore the missing values and their effects in order to avoid the need of multiple passes over the data that methods like median polish require. As pointed out in [13, 14], this can have a dramatic effect on the calculation of effects. Our method chooses to go over more iterations and make each iteration less I/O costly (by estimating data instead of reading the actual data cell values).

6 Conclusions

In this paper we have described an approximate method to do EDA, using the median polish procedure over datacubes. We have shown that the method scales well with the size of the cube, allowing the user to perform EDA in cases where the cost of doing the standard algorithm would be prohibitive.

The price one pays for using the approximate method is a decrease on the quality of the

β	$CErr$	$AveRSl_1$	$AveRSl_2$	$AveRSl_3$	$AveRSl_4$
0.2	0.0021	0.0742	0.0195	0.0004	0

Figure 13: Relative errors for common effect and average rank slipperage for dataset 3, size 1,000,000 non-zero cells, density 0.95 %

k	Re_1^+	Re_2^+	Re_3^+	Re_1^-	Re_2^-	Re_3^-	Pe_1^+	Pe_2^+	Pe_3^+	Pe_1^-	Pe_2^-	Pe_3^-
10	0.6	1	1	1	1	0.90	0.714	1	1	1	1	0.91
20	0.8	1	1	0.95	0.90	1	0.800	1	1	0.952	0.83	1
30	1	1	1	0.933	1	1	1	1	1	0.938	1	1
40	0.925	0.95	1	0.975	0.95	1	0.851	0.93	1	0.876	0.952	1
50	0.98	1		0.960	1		0.862	1		0.943	1	
60	1	1		0.967	1		1	1		0.909	1	
70	0.971	1		0.986	1		0.875	1		0.897	1	
80	0.975	1		0.988	1		0.941	1		0.988	1	
90	0.977			0.956			0.918			0.918		
100	0.98			1			0.952			1		
110	0.973			0.973			0.866			0.957		
120	1			0.958			0.944			0.96		
130	1			0.977			1			0.95		
140	1			1			1			1		

Figure 14: Positive and negative effect recall and precision values for dataset 3.(k is the size of the set of largest positive or largest negative effects.)

$\parallel R \parallel$	Rec	$Pres$
50	0.46	0.254
100	0.68	0.325
150	0.74	0.462
200	0.81	0.505
250	0.84	0.628
300	0.86	0.682
350	0.911	0.665
400	0.928	0.733
450	0.936	0.781
500	0.930	0.751

Figure 15: Recall and Precision for the residuals of dataset 3 four $\beta = 0.2$.

results. However, as shown in Section 4, the quality of the results obtained by the approximate method is good. In most cases, by selecting the set of large positive and negative effects obtained by the approximate method, one obtains a large percentage of the equivalent set in the standard method. The same is true for the residuals. This has been shown using datasets extracted from real applications.

Although our method performs best when the data is highly clustered in regions of the cube, it was shown that it still outperforms the standard method by a large ratio, when that is not the case. In practice, most datasets will exhibit a high degree of clustering.

We are currently continuing this work by looking at methods to improve the quality of our results (by using different modeling techniques), while retaining the scalability of the method. Also, we want to explore techniques to visualize the results of the EDA. Moreover, we are also examining other EDA and mining techniques that can take advantage of approximate methods.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle*, June 1998.

[2] A. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] D. Barbará. The Role of Approximations in Data Mining Algorithms. Technical report, George Mason University, Information and Software Engineering Department, March 1999.

[4] D. Barbará, W. DuMouchel, C. Faloutsos, P.J. Haas, J.M. Hellerstein, Y. Ioannidis, H.V. Jagadish, T. Johnson, R. Ng, V. Poosala, K.A. Ross, and K.G. Sevcik. The New Jersey Data Reduction Report. *Bulletin of the Technial Committee on Data Engineering*, 20(4):3–45, December 1997.

[5] D. Barbará and M. Sullivan. Quasi-Cubes: Exploiting Approximations in Multidimensional Databases. *SIGMOD Record*, 26(3), September 1997.

[6] D. Barbará and M. Sullivan. Quasi-Cubes: A Space-Efficient Way to Support Approximate Multidimensional Databases. Technical Report ISSE-TR-98-03, George Mason University, Information and Software Engineering Department, October 1998.

[7] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1972.

[8] U.S. Census Bureau. Population data. http://www.census.gov/main/www/access.html.

[9] Arbor Software Coorporation. Application Manager User's Guide. Essbase Version 4.0.

[10] R.J. Earle. Method and Apparatus for Storing and Retrieving Multi-dimensional Data in Computer Memory, October 1994. U.S. Patent No. 5,359,724.

[11] J.E. Gentle. Least absolute values estimation: an introduction. *Communications in Statistics*, B6:313–328, 1977.

[12] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering*, pages 152–159, 1996.

[13] D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Exploring Data Tables, Trends and Shapes.* Wiley, 1985.

[14] D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Understanding Robust and Exploratory Data Analysis.* Wiley, 1986.

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

[16] U.S. Department of Health. Statistics. in http://www.cdc.gov/nchswww/fastats/PDF/461s2t01.pdf, 1995.

[17] E.C. Rhodes. Reducing observations by the method of minimum deviations. *Pilosophical Magazine, 7th Series*, 9:974–992, 1930.

[18] K.A. Ross and D. Srivastava. Fast Computation of Sparse Datacubes. In *Proceedings of the 23rd VLDB Conference, Athens, Greece*, 1997.

[19] S. Sarawagi, R. Agrawal, and N. Meggido. Discovery-driven Exploration of OLAP Data Cubes. In *Proceedings of the International Conference on Extending Data Base Technology*, pages 168–182, 1998.

[20] R.R. Singleton. A method of minimizing the sum of absolute values of deviations. *Annals of Mathematical Statistics*, 11:301–310, 1940.

[21] J.W. Tukey. Data analysis, computation, and mathematics. *Quarterly of Applied Mathematics*, 30:51–65, 1972.

[22] J.W. Tukey. *Exploratory Data Analysis.* Addison Wesley, 1977.