Scalable Informative Rule Mining

Guoyao Feng
Carnegie Mellon University
gfeng@andrew.cmu.edu

Lukasz Golab
University of Waterloo
lgolab@uwaterloo.ca

Divesh Srivastava
AT&T Labs - Research
divesh@research.att.com

***Abstract*—We present SIRUM: a system for Scalable Informative RUle Mining from multi-dimensional data. Informative rules have recently been studied in several contexts, including data summarization, data cube exploration and data quality. The objective is to produce a small set of rules (patterns) over the values of the dimension attributes that provide the most information about the distribution of a numeric measure attribute. Within SIRUM, we propose several optimizations for *tall*, *wide* and *distributed* datasets. We implemented SIRUM in Spark and observed significant performance and scalability improvements on real datasets due to our optimizations. As a result, SIRUM is able to generate informative rules on much wider and taller datasets than using distributed implementations of the previous state of the art.**

I. Introduction

In this paper, we describe our SIRUM system for Scalable Informative RUle Mining from tall, wide and distributed data. The input is a dataset with a number of categorical dimension attributes and a numeric measure attribute m. The output is a list of rules, represented as conjunctions of values of the dimension attributes, that provide the most information about the distribution of m in the given dataset.

Example: Consider the flight dataset shown in Table I; ignore the last three columns labeled $\hat{m}_i$ for now. For each flight, the dataset includes the day of the week, origin and destination airports, and a measure attribute "Late" denoting how late the flight was. Suppose a data analyst wants to summarize the distribution of flight delays for different combinations of values of the dimension attributes, perhaps as a precursor to buliding a prediction model. SIRUM outputs the rules shown in Table II, r_1 through r_4. Each rule is a tuple from the *data cube* over the three dimension attributes ("*" corresponds to all possible values), with the expected (average) value of the Late attribute for each rule and the count of matching tuples (frequency) shown in the last two columns. The first rule, r_1, states that the 14 flights in this dataset were late by 10.4 minutes on average. The second rule states that the four London-bound flights were late by 15.3 minutes on average. As we will formalize shortly, this rule provides the most information about the Late attribute beyond what we know so far from the first rule. Intuitively, this is because the average delay of London-bound flights is significantly different than the average delay in the entire dataset and there are many London-bound flights. Given the information from the first two rules, the next most informative rule, r_3, states that the two flights departing on a Friday were 18 minutes late on average, and so on.

In addition to data profiling and summarization [1], [2], informative rule mining enables smart data cube exploration [3]. Returning to the flight dataset, suppose the analyst already knows the average flight delay in the entire dataset and the average delay of flights out of SF. This information corresponds to the first two rows in Table III. SIRUM recommends the two rules shown at the bottom of Table III, namely (*, *, London) and (Fri, London, LA), as they provide the most additional information about the distribution of flight delays. The analyst can then drill down and examine the records corresponding to these two rules. Furthermore, informative rule mining is useful in diagnosing data quality issues (see, e.g., Data X-Ray [4] and Data Auditor [5]). In this problem, the measure attribute denotes the quality of a tuple (e.g., the number of clean attributes of a tuples) and the goal is to determine if data quality problems are correlated with certain values of the dimension attributes. Here, SIRUM can identify subsets of the data with an unusually high or low number of dirty records.

TABLE I: A flight dataset

id	Day	Origin	Dest.	Late	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$
1	Fri	SF	London	20	10.4	15.3	22.4
2	Fri	London	LA	16	10.4	8.4	13.6
3	Sun	Tokyo	Frankfurt	10	10.4	8.4	7.8
4	Sun	Chicago	London	15	10.4	15.3	12.9
5	Sat	Beijing	Frankfurt	13	10.4	8.4	7.8
6	Sat	Frankfurt	London	19	10.4	15.3	12.9
7	Tue	Chicago	LA	5	10.4	8.4	7.8
8	Wed	London	Chicago	6	10.4	8.4	7.8
9	Thu	SF	Frankfurt	15	10.4	8.4	7.8
10	Mon	Beijing	SF	4	10.4	8.4	7.8
11	Mon	SF	London	7	10.4	15.3	12.9
12	Mon	SF	Frankfurt	5	10.4	8.4	7.8
13	Mon	Tokyo	Beijing	6	10.4	8.4	7.8
14	Mon	Frankfurt	Tokyo	4	10.4	8.4	7.8

TABLE II: Four informative rules over the flight dataset

Rule	Day	Origin	Dest.	expected_Late	frequency
r_1	*	*	*	10.4	14
r_2	*	*	London	15.3	4
r_3	Fri	*	*	18	2
r_4	Sat	*	*	16	2

Informative rule mining is *interactive*: a user may request a list of informative rules, explore the data, and ask for more rules (for the same or different measure attribute). This motivates the need for efficiency. However, challenges arise when computing informative rules over big data, both in terms of the number of entities and in terms of the number

TABLE III: Two tuples corresponding to a user's prior knowledge, followed by two informative rules, over the flight dataset

Day	Origin	Dest.	expected_Late	frequency
*	*	*	10.4	14
*	SF	*	11.8	4
*	*	London	15.3	4
Fri	London	LA	16	1

of dimension attributes. For example, a flight data set may contain millions of flights, with many dimension attributes corresponding to the variety of information collected by airport processes and by scanning passengers' boarding passes at various points: weather, queues at the origin airport, runway congestion, connecting flights, etc. This means that the input is *tall* (many rows) and *wide* (many columns), leading to a very large number of candidate rules, which grows exponentially with the number of dimension attributes and may be larger than the dataset itself.

Big data are typically processed using distributed computing platforms. Efficiency matters not only to enable interactive applications, but also to save money if the computation is done on the cloud (e.g., using Amazon EC2 or Microsoft Azure) and costs are incurred based on resource utilization. However, distributed generation of informative rules is challenging for several reasons. It is an iterative process, which repeatedly selects the next most informative rule (details in Section II). This requires more careful optimization than traditional batch processing, and may incur high disk I/O to repeatedly scan the input. Additionally, the very large number of intermediate results (possible rules) may cause CPU and I/O overhead.

Despite the recent interest in informative rule mining [1], [2], [3], existing techniques do not scale to tall and wide datasets: they require multiple scans of large datasets and do not leverage parallel computation. Moreover, there are no distributed algorithms for informative rule mining that can work with data stored in a distributed file system in-situ. These are exactly the issues we address with SIRUM. The contributions of this paper are as follows:

1) SIRUM: a *distributed* framework for Scalable Informative RUle Mining. We implemented SIRUM using Spark [6], a main-memory platform for parallel iterative data processing, with data stored in the Hadoop Distributed File System (HDFS). We justify our choice of Spark in Section VI by comparing it to versions of SIRUM implemented using a traditional MapReduce-based framework (Hive), SparkSQL and PostgreSQL.
2) A profiling study of a distributed implementation of the state of the art, which reveals the bottlenecks of informative rule generation from tall and wide tables, suggesting that parallelism alone is insufficient and algorithmic improvements are needed.
3) Algorithmic optimizations for distributed informative rule generation from tall and wide tables. For tall tables, we optimize away table scans to minimize data shuffling and disk I/O. For wide tables, we heavily prune the space of candidate rules to reduce CPU and memory usage while maintaining good quality of the solution.

TABLE IV: Symbols used in this paper

Symbol	Meaning
D	A dataset
s	A random sample from D
$\lvert s\rvert$	The size (number of tuples) of s
A_j	A dimension attribute
$dom(A_j)$	The active domain of A_j
d	The number of dimension attributes
m	A measure attribute
$\hat{m}$	An estimate of m
t	A tuple
$t[A_j]$	The value of the A_j attribute of tuple t
r	A rule
$m(r)$	Average value of $t[m]$ of the tuples matching r
$\hat{m}(r)$	Average value of $t[\hat{m}]$ of the tuples matching r
$\lambda(r)$	Multiplier of r used during iterative scaling
R	The list of rules already generated
k	The number of rules to be generated

In the remainder of this paper, Section II presents background on informative rule mining; Section III outlines a baseline implementation of SIRUM and profiles its performance on real datasets; Sections IV and V present our performance optimizations for wide and tall tables, respectively; Section VI explains our experimental findings; Section VII discusses prior work; and Section VIII concludes the paper.

II. Background

We begin by describing a general version of the informative rule generation process used in prior work [1], [2], [3]. Table IV defines important symbols.

We are given an integer k, and a dataset D with d dimension attributes, A_1 through A_d, and a numeric measure attribute m. The output is a list of k rules. Since the rules are meant to be interpretable, k is expected to be small, say, between 10 and 50. A rule r is a tuple from $dom(A_1) \cup \{*\} \times \cdots \times dom(A_d) \cup \{*\}$, i.e., from the data cube over the dimension attributes. A tuple t matches r, denoted by $t \asymp r$, if $r[A_j] =$ '*' or $t[A_j] = r[A_j]$ for each dimension attribute A_j. For example, tuple 4 from Table I matches rules r_1 and r_2 from Table II, but not r_3 or r_4.

Let $|r|$ be the number of tuples matching rule r, i.e., $|r| = |\{t \in D, t \asymp r\}|$. Let $m(r) = \frac{1}{|r|}\sum_{t\in D, t\asymp r} t[m]$, i.e., the average value of the measure attribute m of the tuples that match r. Note that the last two columns of Table II, labeled "expected_Late" and "frequency", show $m(r)$ and $|r|$, respectively, for each rule.

Since informative rule mining is a hard problem [1], we follow previous work and use a simple greedy heuristic. We first select the all-stars rule that matches all tuples. Next, we iterate $k-1$ times, each time selecting the next best rule that contains the most additional information. We describe the details of selecting the next best rule below.

Let $t[\hat{m}]$ be the estimated value of $t[m]$ based on the rules generated so far. Let $\hat{m}(r) = \frac{1}{|r|}\sum_{t\in D, t\asymp r} t[\hat{m}]$, the average value of the estimate $\hat{m}$ of the tuples that match a rule r.

Algorithm 1 Iterative Scaling

```
Input: $D, R, \lambda, m(r)$ for all $r \in R$, $\epsilon$
 1: $DIFF \leftarrow ARRAY(|R|, 0)$
 2: while true do
 3:     for $i$ 1 to $|R|$ do
 4:         $\hat{m}(r_i) \leftarrow \frac{1}{|r_i|} \sum_{t \in D, t \asymp r_i} t[\hat{m}]$
 5:         $DIFF[i] \leftarrow |m(r_i) - \hat{m}(r_i)|/|m(r_i)|$
 6:     end for
 7:     $next \leftarrow \underset{i}{\text{argmax}} DIFF[i]$
 8:     if $DIFF[next] > \epsilon$ then
 9:         $\lambda(r_{next}) \leftarrow \lambda(r_{next}) * \frac{m(r_{next})}{\hat{m}(r_{next})}$
10:         for each $t \asymp r_{next}$, set $t[\hat{m}] = \prod_{r, t \asymp r} \lambda(r)$
11:     else
12:         break
13:     end if
14: end while
```

We use the principle of *maximum entropy* to determine the estimates: we set the $t[\hat{m}]$s to maximize $-\sum_{t \in D} t[\hat{m}] \log t[\hat{m}]$ while agreeing with what each rule reveals about the tuples that match it, namely $m(r) = \hat{m}(r)$. Intuitively, maximizing entropy makes no assumptions about the distribution of m aside from the information provided by the rules generated so far.

Returning to Table I, the column labeled $\hat{m}_1$ shows the estimates of the Late attribute based on the first rule. At this point, all we know is that the average value of Late is 10.4 in the entire dataset, so the maximum-entropy solution is to set each $t[\hat{m}]$ to 10.4. After adding r_2 (see column $\hat{m}_2$), t_1, t_4, t_6 and t_{11} must have $t[\hat{m}] = 15.3$ to satisfy $m(r_2) = \hat{m}(r_2)$. This means that the maximum-entropy solution is to set the other $t[\hat{m}]$'s to 8.4 each to satisfy $m(r_1) = \hat{m}(r_1) = 10.4$. Similarly, the column labeled $\hat{m}_3$ shows the maximum-entropy solution for the first three rules.

Following recent work on informative rule mining [1], we solve the maximum-entropy optimization problem using *iterative scaling* [7], [8]. For each tuple t, we set $t[\hat{m}] = \prod_{r, t \asymp r} \lambda(r)$, where $\lambda(r)$ is a *multiplier* associated with a rule r [9]. Whenever a new rule r is added, we set its $\lambda(r)$ to one by default, we compute its $m(r)$, and we run Algorithm 1 on R, the list of rules generated so far. Assume that the estimates $t[\hat{m}]$ are all set to one at the beginning, before generating the first rule. Line 1 of the algorithm initializes a DIFF array to all zeros. In lines 2 through 6, we compute the current $\hat{m}(r)$ values for each rule, as well as their differences from the actual $m(r)$s (DIFF). In line 7, we find the rule with the greatest DIFF, and, if DIFF is greater than some small number ϵ (say, 0.01), we update λ for this rule as shown in line 9. We must then update the $t[\hat{m}]$ values of all tuples matching this rule (line 10). We keep going until all the rules have DIFFs below ϵ, i.e., until the $\hat{m}(r)$s have (nearly) converged to $m(r)$s.

Returning to the flight example, when r_1 is generated, $\lambda(r_1) = 1$ and $t[\hat{m}] = 1$ for all tuples by default, and therefore $\hat{m}(r_1) = 1$. Since $m(r_1) = 10.4$, we set $\lambda(r_1) = 1 * 10.4/1 = 10.4$ and update all the $t[\hat{m}]$ to 10.4 in line 11. At this point, $m(r_1) = \hat{m}(r_1)$ and iterative scaling stops. After r_2 is generated, $\lambda(r_2) = 1$ by default and we update $\lambda(r_2) = \lambda(r_2) * \frac{m(r_2)}{\hat{m}(r_2)} = 1 * \frac{15.3}{10.4} = 1.47$. This means that $t[\hat{m}]$ for t_1, t_4, t_6, and t_{11} must be updated to 15.3: for these tuples, $t[\hat{m}] = \lambda(r_1) * \lambda(r_2)$. It causes $\hat{m}(r_1)$ to change to 11.76 and therefore now $m(r_1) \neq \hat{m}(r_1)$. So, we scale $\lambda(r_1)$ by setting it to $\lambda(r_1) * \frac{m(r_1)}{\hat{m}(r_1)} = 10.4 * 10.4/11.76 = 9.2$. Note that t_1, t_4, t_6, and t_{11} now have $t[\hat{m}] = \lambda(r_1) * \lambda(r_2) = 13.5$, which causes $\hat{m}(r_2)$ to change to 13.5. Now, since $m(r_2) \neq \hat{m}(r_2)$, we set $\lambda(r_2) = 1.47 * 15.25/13.5 = 1.67$. After several more iterations, we settle on $\lambda(r_1) = 8.4$ and $\lambda(r_2) = 1.8$, which gives the estimates shown in the $\hat{m}_2$ column of Table I.

We now explain the notion of informativeness. We want to minimize the Kullback-Leibler (KL) divergence between the true $t[m]$ values and their estimates $t[\hat{m}]$ calculated from the maximum-entropy distribution induced by the rules:

$$KL(m||\hat{m}) = \sum_{t \in D} t[m] \log \frac{t[m]}{t[\hat{m}]}.$$

(Technically, KL divergence measures the distance between two distributions, so we need to normalize the $t[m]$ and $\hat{t}[m]$ values so they add up to one.)

In the context of the greedy heuristic for rule generation, the next best rule is the one that gives the greatest reduction in KL divergence. However, to find this rule, for each candidate r, we would have to compute the KL divergence assuming that r has been selected, which requires iterative scaling. Instead, we calculate an upper bound on the reduction in KL divergence of a rule r, referred to as *gain* [1], [3].

$$gain(r) = \sum_{t \in D, t \asymp r} t[m] \log \frac{\sum_{t \in D, t \asymp r} t[m]}{\sum_{t \in D, t \asymp r} t[\hat{m}]}.$$

For instance, in our flight example, after r_1 has been added, (*,*,London) is selected next because it has the highest gain.

To summarize the informative rule generation process, we first select the all-stars rule and run iterative scaling. Then, we iterate $k - 1$ times, once for each remaining rule to be generated, and perform two steps in each iteration: 1) compute the gain of each candidate rule and add to R the one with the highest gain, and 2) run iterative scaling.

III. Baseline Implementation and Profiling

In this section, we evaluate the performance of Baseline version of SIRUM, which corresponds to a distributed implementation of the current state of the art, plus some additional straightforward optimizations. Our results will reveal the bottlenecks that we address in the optimized version of SIRUM described in Sections IV and V.

A. Baseline SIRUM

We implemented Baseline (and optimized) SIRUM using Spark [6], with the input dataset D stored in HDFS. Spark is a distributed computing platform that allows caching distributed data sets in the memory of a cluster. This functionality, called Resilient Distributed Datasets (RDD), allows us to load D (plus the estimates $t[\hat{m}]$) into memory and repeatedly scan it (when finding the next most informative rule and running iterative scaling) without the overhead of HDFS I/O.

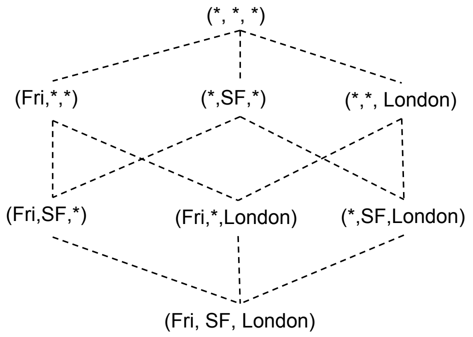

Fig. 1: Cube lattice for (Fri, SF, London)

SIRUM implements the greedy heuristic described in the previous section: in each iteration, we 1) compute the gain of each candidate rule and add to R the one with the highest gain, and 2) run iterative scaling. We implemented the first step using a distributed data cube algorithm from [10], with the dimension attributes of D as the group-by attributes, and SUM(m) and SUM($\hat{m}$) as the aggregate functions[1]. Each tuple in the data cube corresponds to a possible rule and its aggregate values can be used to compute the gain of the corresponding rule as $SUM(m) \cdot \log \frac{SUM(m)}{SUM(\hat{m})}$.

Before describing the data cube algorithm, we review the concepts of *cube lattice*, *ancestors* and *descendants*. In Figure 1, we show the cube lattice for the dimension attributes of the first tuple from Table I, (Fri, SF, London). The elements of the cube lattice are the tuples in the data cube to which (Fri, SF, London) contributes. The bottom level of the cube lattice is the tuple itself, the next level up corresponds to the elements with exactly one star, and so on up to the dth level, where d is the number of dimension attributes. Elements in higher levels are *ancestors* of those from lower levels which are connected to them. For example, (*,*,*) is the only ancestor of (*,SF,*). Similarly, elements in lower levels are *descendants* of those from higher levels which are connected to them. For example, (Fri, SF, London) is the only descendant of (Fri, SF, *). The cube lattice for multiple tuples combines the lattices of the individual tuples, merging every common ancestor into a single node in the combined lattice.

In the map stage of the distributed data cube algorithm, each mapper processes the part of D that is locally stored in its memory. For each such tuple t, the mapper emits a key-value pair for every element in t's cube lattice including t itself, with the key being the element and the value being a pair ($t[m]$, $t[\hat{m}]$). The output is then shuffled and assigned to the reducers. Each reducer receives a subset of the keys (possible rules) to process, and, for each key, it computes the gain from the partially-aggregated values produced by the mappers.

After choosing the most informative rule, we run iterative scaling (Algorithm 1). In each round of iterative scaling, we join D with R, the list of rules generated so far, to re-compute lines 3-6, and scan D again in line 10 to update the estimates $\hat{m}$ of affected tuples.

[1]We also compute $AVG(m)$ and count(*) at the same time since we will need to include these with the selected rules.

The number of candidate rules can be very large. Existing pruning techniques draw a small random sample s from D and only consider rules which appear in the sample [1]. The idea is that rules with frequently-occurring combinations of dimension attribute values are likely to appear in a sample and are more likely to have high gain. We refer to this method as *sample-based* candidate pruning, in contrast to *exhaustive* candidate selection described above, which computes the gain of every possible rule.

Suppose we sample two tuples from Table I and we get tuples 4 and 9. Projected onto their dimension attributes, these are (Sun, Chicago, London) and (Thu, SF, Frankfurt). For each tuple in the sample, sample-based pruning computes the *Lowest Common Ancestor* (LCA) of it and each tuple in D, where non-matching attribute values are replaced with stars. For example, the LCA of (Sun, Chicago, London) and (Fri, SF, London) is (*,*,London). Using tuples 4 and 9 as the sample, we get the following LCAs: (*,*,*), (*,*,London), (*,*,Frankfurt), (*,Chicago,*), (*, SF, *), (Sun,*,*), (*,SF, Frankfurt), (Sun, Chicago, London) and (Thu, SF, Frankfurt). Next, it generates all the ancestors of each LCA, and uses only the LCAs and their ancestors as candidate rules. In our example, the candidate set is: (*,*,*), (*,*,London), (*,*,Frankfurt), (*,Chicago,*), (*, SF, *), (Sun,*,*), (Thu,*,*), (Sun,Chicago,*), (Sun,*,London), (*,Chicago,London), (Thu,SF,*), (Thu,*,Frankfurt), (*,SF, Frankfurt), (Sun, Chicago, London) and (Thu, SF, Frankfurt). This gives only 15 candidate rules compared to 73 possible rules in the full datacube.

It has been shown in recent work [1] that sample-based pruning significantly improves efficiency with only a small penalty in KL-divergence. We incorporate sample-based pruning in Baseline SIRUM as follows. Rather than computing the full data cube to obtain the gain of each possible rule, we start by drawing a random sample s from D and computing a cross-product of s and D to obtain the LCAs. The LCAs are stored in an RDD. Next, we emit the ancestors of each LCA (and the LCA itself) in the map stage, and compute the information gain of the LCAs and their ancestors in the reduce stage.

We now describe a straightforward improvement. Since R and s are much smaller than D, we can use Spark's *broadcast join* [11], [12], i.e., map-side join, to compute the LCAs (which requires s join D) and to compute lines 4-6 of the iterative scaling algorithm (which requires R join D). We replicate the small datasets to each mapper and keep them in memory as *broadcast variables*. This eliminates the need to re-partition the join inputs on the join attribute, and instead, each mapper can compute the join over its partition of D locally.

To summarize, Baseline SIRUM reflects our best effort to produce an efficient distributed implementation of the previous state-of-the-art, including distributed data cube computation, sample-based candidate pruning and Spark's broadcast join.

B. Profiling Baseline SIRUM

We now measure the runtime of the two steps performed by Baseline SIRUM in each iteration: 1) generating candidate rules using sample-based pruning, computing their gain (in parallel) and selecting the most informative one, and 2) iterative

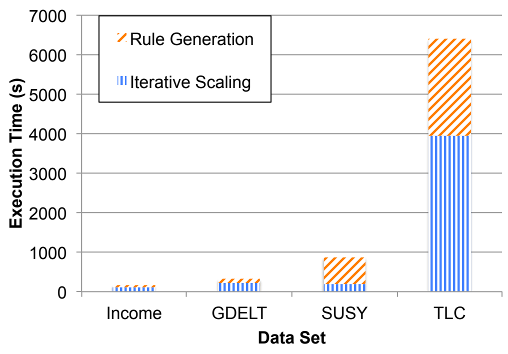

Fig. 2: Baseline SIRUM runtimes broken into rule generation and iterative scaling (k = 10, $|s|$ = 64)

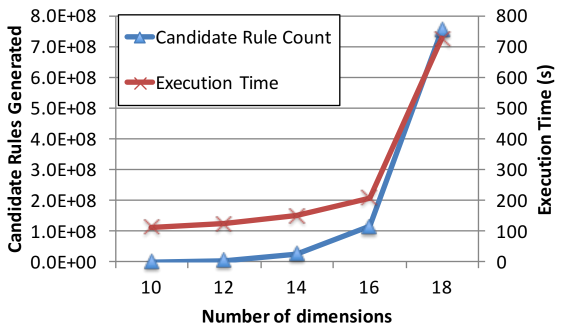

Fig. 3: Rule generation running time and number of candidate rules versus the number of dimension attributes (k = 10, $|s|$ = 64, SUSY)

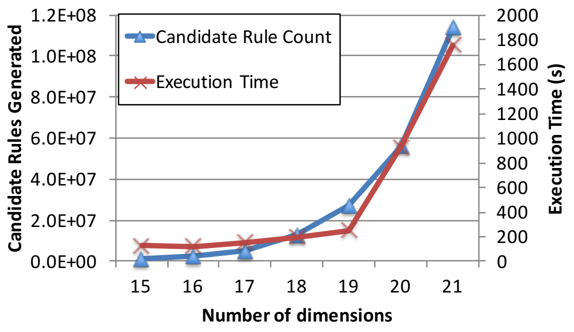

Fig. 4: Rule generation running time and number of candidate rules versus the number of dimension attributes (k = 10, $|s|$ = 64, ONP)

scaling. We show that the first step is the bottleneck for wide datasets and the second step is the bottleneck for tall datasets.

Our experiments are performed on a 16-node cluster using four real datasets: Income, GDELT, SUSY and TLC. We provide details in Section VI-A, but, in brief: Income has roughly 1.5 million tuples and 9 dimension attributes; GDELT has 3.8 million tuples and 9 dimension attributes; SUSY has 5 million tuples and 18 dimension attributes; and TLC has 160 million tuples and 9 dimension attributes.

Figure 2 shows the runtimes of the two steps for each dataset, assuming $k = 10$ and $|s| = 64$, where $|s|$ is the size of the sample used in sample-based candidate pruning; other values of these parameters led to similar conclusions. Rule generation is the bottleneck for the widest dataset, SUSY, and a large part of the runtime for the tallest dataset, TLC. Iterative scaling is the bottleneck for Income, GDELT and to a lesser extent TLC. Furthermore, the total runtime is by far the highest for TLC, which is larger than the main memory of the cluster, and therefore causes disk I/O during rule generation and each round of iterative scaling.

Next, we create projections of SUSY on the first 10 through 18 attributes. Figure 3 plots the runtime of rule generation and the number of candidate rules considered. Despite using sample-based candidate pruning, the number of candidate rules and the running time of rule generation blow up at 18 attributes. This leads to high CPU and memory usage of the mappers, which generate LCAs and their ancestors, resulting in stragglers, additional CPU overhead due to garbage collection to free up memory, and even disk I/O.

The data cube algorithm used by Baseline SIRUM computes aggregates for each candidate rule in a single map-reduce round. Thus, one may argue that a way to limit the CPU and memory requirements is to split the processing of candidate rules into multiple map-reduce rounds, each round processing a subset of the cube lattice. This way, we reduce the load on the mappers in each round, but of course we incur the overhead of starting up new map-reduce operations. We implemented such a multi-round algorithm using ideas similar to those from traditional hash-based data cube algorithms (see, e.g., [13]). Figure 4 shows the runtime of rule generation and the number of candidate rules considered for an even wider dataset, ONP, which has 58 dimension attributes and 48 thousand rows (details in Section VI-A). We created projections of ONP on the first 15 to 21 attributes and observed a blow-up at 21 attributes. Baseline SIRUM with the multi-round data cube algorithm did not terminate in reasonable time for 22 or more attributes.

From these results, we conclude that even an optimized distributed implementation of the previous state-of-the-art does not scale to wide and tall tables due to the size of the candidate rule space and the need to repeatedly scan the input.

IV. Optimizing for Wide Datasets

As Figure 3 and Figure 4 suggest, when the number of dimension attributes d exceeds a certain threshold d_0, whose value depends on the system resources and on the dataset, the rule generation process used by Baseline SIRUM becomes the major bottleneck. This is because the number of candidate rules grows exponentially with d, even with sample-based candidate pruning. In this section, we present a new candidate pruning technique that addresses this problem. In the next section, we will optimize iterative scaling, which is a bottleneck for tall datasets.

Before we present our solution, we comment on the difficulty of pruning the search space of informative rules. As observed in [1], Apriori-like optimizations are not applicable. A candidate rule that is not informative may become informative in subsequent iterations after several rules have already been

Algorithm 2 Generating one rule using Column Partitioning

```
Input: $D, d, s, d_0, p$
 1: $LCA = D \times s$
 2: $Bestrules = \emptyset$
 3: $\theta = \lceil \frac{d}{d_0} \rceil$
 4: for i = 1 to p do
 5:     Randomly split the $d$ dimension attributes of $D$ into
          $\theta$ groups of up to $d_0$ columns per group
 6:     for j = 1 to $\theta$ do
 7:         $ProjLCA$ = $LCA$ projected onto the columns of
              the $j$th group
 8:         $Bestrules = Bestrules \cup$ a rule from $ProjLCA$
              plus their ancestors having the highest gain
 9:     end for
10: end for
11: Return a rule from $Bestrules$ with the highest gain
```

selected. Furthermore, while the number of tuples matching a rule cannot be larger than the number of tuples matching any of its descendants (recall Section III-A), its information gain may be higher or lower than that of its descendants. Thus, top-down pruning, starting from $(*, *, \ldots, *)$ and traversing the lattice downwards, based on the number of tuples matching a rule (analogous to support in association rules) is not effective. We have developed and tested several simple top-down pruning rules that bound the maximum gain of any descendant within a particular sub-lattice. However, in terms of the tradeoff between runtime and the KL-divergence of the resulting rules, we were unable to beat the bottom-up pruning strategy that we describe next (and we defer further investigation of top-down pruning to future work).

The technique we included in SIRUM, referred to as Column Partitioning (CP), is summarized in Algorithm 2. We only show how to select a single rule; the process must be repeated k times, with iterative scaling performed after each new rule is selected, to select k rules. Suppose we empirically observe that Baseline SIRUM struggles with more than d_0 dimension attributes. First, we draw a random sample s from D and compute a cross-product of s and D to obtain the LCAs, as before in sample-based candidate pruning (Line 1). Then, we randomly split the dimension attributes into $\theta = \lceil \frac{d}{d_0} \rceil$ column groups (Line 5). Next, for each group, 1) we project the LCAs onto the dimension attributes belonging to this group (Line 7), setting all other attribute values to stars; 2) we only compute the gain of the projected LCAs and their ancestors; and 3) we add to $Bestrules$ a rule with the highest gain (Line 8). Note that gain computation (Line 8) can proceed in parallel for each column group. Finally, we repeat this process on p random partitions of the dimension attributes (Lines 4-10) and select a rule with the highest gain from the $p \times \theta$ rules in $Bestrules$ (Line 11).

Figure 5 illustrates an example column partitioning. Let A_1 through A_{16} be the 16 dimension attributes, as illustrated on the first line. Assume $d_0 = 4$, and therefore $\theta = 4$ and each part (group) consists of four attributes. The next line, labeled "Random Partition Scheme 1", shows a possible partitioning of the dimension attributes. For example, the candidate rules arising from the first group consist of LCAs with non-stars A_5, A_{10}, A_{11} or A_{16} and stars elsewhere, and their ancestors.

$(A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16})$

Random Partition Scheme 1: $(A_5, A_{10}, A_{11}, A_{16})$, $(A_6, A_7, A_{12}, A_{15})$, $\mathbf{(A_2, A_3, A_8, A_9)}$, $(A_1, A_4, A_{13}, A_{14})$

Random Partition Scheme 2: $(A_7, A_{13}, A_{11}, A_{16})$, $\mathbf{(A_8, A_5, A_2, A_{15})}$, (A_{12}, A_3, A_6, A_1), $(A_9, A_4, A_{10}, A_{14})$

Random Partition Scheme 3: $(A_1, A_{13}, A_{15}, A_2)$, $(A_8, A_5, A_{16}, A_{11})$, $\mathbf{(A_{12}, A_4, A_6, A_7)}$, $\mathbf{(A_9, A_3, A_{10}, A_{14})}$

LCA: $(V_1,*,V_3,V_4,*,*,V_7,*,V_9,*,V_{11},V_{12},*,V_{14},V_{15},*)$

Projected to $\mathbf{(A_2, A_3, A_8, A_9)}$: $(*,*,V_3,*,*,*,*,*,V_9,*,*,*,*,*,*,*)$

Projected to $\mathbf{(A_8, A_5, A_2, A_{15})}$: $(*,*,*,*,*,*,*,*,*,*,*,*,*,*,V_{15},*)$

Projected to $\mathbf{(A_{12}, A_4, A_6, A_7)}$: $(*,*,*,V_4,*,*,V_7,*,*,*,*,V_{12},*,*,*,*)$

Projected to $\mathbf{(A_9, A_3, A_{10}, A_{14})}$: $(*,*,V_3,*,*,*,*,*,V_9,*,*,*,*,V_{14},*,*)$

Fig. 5: Example of Column Partitioning, assuming 16 attributes and three partitioning schemes with 4 attributes per part. A particular LCA is illustrated at the bottom, along with its projection to the three bolded groups.

Figure 5 shows $p = 3$ random partition schemes, leading to $p \times \theta = 12$ possibly overlapping regions of the search space. Now consider the four column groups highlighted in bold: one each from the first two partition schemes and two from the last one. Suppose one of the LCAs generated from the sample has stars everywhere except A_1, A_3, A_4, A_7, A_9, A_{11}, A_{12}, A_{14} and A_{15}, as illustrated (where V_i is some value from attribute A_i). When projected onto the attributes of the first bolded group, this LCA has a_3, a_9 and stars elsewhere. Projections onto other bolded groups are also illustrated.

The intuition behind column partitioning is that informative rules should not have too many constants: a rule with many constants likely matches few tuples and will not have high gain unless all the tuples that match it have very inaccurate estimates $t[\hat{m}]$. Thus, since the search space is too large, we should focus on subspaces with rules having few constants. In particular, since each of the θ parts has at most d_0 attributes, column partitioning amounts to exploring $p \times \theta$ manageable regions, of size up to $O(\frac{d}{d_0} \times 2^{d_0} \times |s|)$, of a prohibitively large candidate rule space. Of course, this only considers rules with at most d_0 constants, all from the same part, and stars elsewhere.

Column partitioning requires two parameters: d_0, the maximum number of attributes per group, and p, the number of random partitioning schemes to consider. We can determine d_0 by running SIRUM in "exploratory" mode whenever a new dataset is uploaded to see how many dimension attributes it can handle in reasonable time. We will experimentally examine the effect p in Section VI.

V. Optimizing for Tall Datasets

In Section IV, we presented a technique for pruning the space of candidate rules, which, as we will show, helps SIRUM scale with the number of attributes. In this section, we present additional and complementary techniques for tall datasets: optimizing away table scans during iterative scaling

TABLE V: RCT after the third rule has been generated

BA	count	$SUM(t[m])$	$SUM(t[\hat{m}])$
1000	9	68	75.6
1100	3	41	45.9
1010	1	16	8.4
1110	1	20	15.3

(Section V-A) and sampling the input to fit the available main memory and avoid repeated disk I/O (Section V-B).

A. *Fast Iterative Scaling*

Recall Algorithm 1 for iterative scaling, which runs whenever we select a new rule. In our experiments with Baseline SIRUM, iterative scaling looped ten or more times until the multipliers converged. Every time, it accessed D twice: to compute the $\hat{m}(r_i)$s in lines 2-6 and to update the $t[\hat{m}]$s in line 10. Furthermore, when accessing D, the $t \asymp r$ operation is executed for every tuple, which compares each of t's attribute values to those of r; this is done even for rules that have been added previously. Our improved iterative scaling technique is shown in Algorithm 3 and described below.

First, we cache the results of $t \asymp r$ for previously added rules. We maintain a bit array BA for each tuple t, whose ith entry, $t.BA[i]$, is one if $t \asymp r_i$ and zero otherwise. For example, in Table I, t_1 has $t.BA = 1100$ because it matches r_1 and r_2 from Table II. We also create a bit array for each rule r_i, $r_i.BA$, which has a one in the ith entry and zeros elsewhere. When a new rule r_w is added to R, we update the BA values of each tuple by testing $t \asymp r_w$ (lines 1-5 in Algorithm 3). Afterwards, $t \asymp r_i$ operations become a bitwise AND of $t.BA$ and $r_i.BA$.

Next, we avoid repeatedly accessing D during iterative scaling. Recall Table I and suppose that r_3 has just been generated. The first step is to set $t.BA[3] = 1$ for all $t \asymp r_3$. The first two bits of the BA have already been set when the r_1 and r_2 were generated. Next, we compute $count(*)$, $SUM(t[m])$ and $SUM(t[\hat{m}])$ grouped-by BA (line 6 in Algorithm 3). The result, which we refer to as a Rule Coverage Table (RCT), is shown in Table V. Note that the $t[\hat{m}]$ values used as input to the above group-by query are as shown in the column $\hat{m}_2$ in Table I. Also, note that the first bit of every BA is one since every tuple matches the all-stars rule.

Each row of the RCT in Table V describes a subset of tuples from D that is pairwise disjoint with every other row of the RCT, as illustrated in Figure 6. Each subset is uniquely defined by the set of rules matched by the tuples inside. For example, $BA = 1010$ corresponds to tuples that match only r_1 and r_3, i.e., tuple 2. A key observation is that all tuples in the same subset share the same estimates; e.g., any tuple with $BA = 1010$ has the same $t[\hat{m}] = \lambda(r_1) * \lambda(r_3)$. This property allows the RCT to keep a minimal amount of information necessary for iterative scaling.

The RCT maintains pre-aggregated $SUM(t[\hat{m}])$ values, making it easy to compute the $\hat{m}(r_i)$s by merging the partial aggregates. To compute $\hat{m}(r_i)$, we find all the rows in the RCT that have the ith bit of BA set to one, and divide the sum of

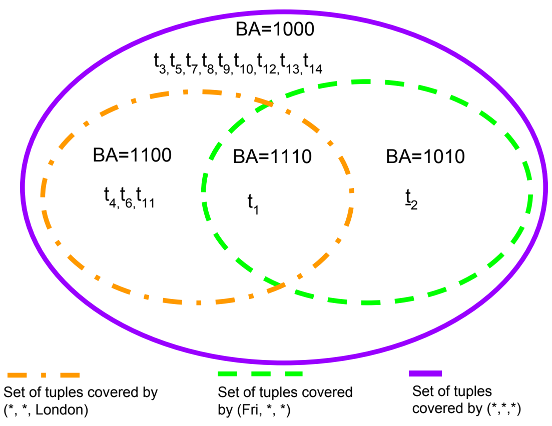
BA=1000
$t_3, t_5, t_7, t_8, t_9, t_{10}, t_{12}, t_{13}, t_{14}$
BA=1100
t_4, t_6, t_{11}
BA=1110
t_1
BA=1010
t_2
Set of tuples covered by (*, *, London)
Set of tuples covered by (Fri, *, *)
Set of tuples covered by (*,*,*)

Fig. 6: Illustration of the RCT from Table V

Algorithm 3 Iterative Scaling with RCT

```
Input: $D, R, \lambda$, $m(r)$ for all $r \in R$, $\epsilon$, newly added rule $r_w$
1: for $t \in D$ do
2:     if $t \asymp r_w$ then
3:         $t.BA[w] \leftarrow 1$                # $t.BA[w]$ default to 0
4:     end if
5: end for
6: Group by $t.BA$ and aggregate over $COUNT(*)$, $SUM(t[m])$, and $SUM(t[\hat{m}])$ to compute the $RCT$
7: while true do
8:     $DIFF \leftarrow ARRAY(|R|, 0)$
9:     for $r_i \in R$ do
10:        $\hat{m}(r_i) \leftarrow \frac{\sum_{p \in RCT, r_i.BA \& p.BA \neq 0} p.SUM(t[\hat{m}])}{\sum_{p \in RCT, r_i.BA \& p.BA \neq 0} p.count}$
11:        $DIFF[i] \leftarrow |m(r_i) - \hat{m}(r_i)|/|m(r_i)|$
12:    end for
13:    $next \leftarrow \underset{i}{\text{argmax}} DIFF[i]$
14:    if $DIFF[next] > \epsilon$ then
15:        $\lambda_{old} \leftarrow \lambda(r_{next})$
16:        $\lambda(r_{next}) \leftarrow \lambda(r_{next}) * \frac{m(r_{next})}{\hat{m}(r_{next})}$
17:        for each $p$ in RCT such that $p.BA$ & $r.BA \neq 0$, set $p.SUM(t[\hat{m}]) = p.SUM(t[\hat{m}]) * \frac{\lambda(r_{next})}{\lambda_{old}}$
18:    else
19:        for $t \in D$ do
20:            $t[\hat{m}] \leftarrow \prod_{r \in R, t \asymp r} \lambda(r)$
21:        end for
22:        break
23:    end if
24: end while
```

$SUM(t[\hat{m}])$ by the sum of *count* of these rows (line 11 in Algorithm 3). Then, after scaling $\lambda(r_{next})$ in line 16, we do not access D to modify the affected $t[\hat{m}]$s; instead, we update the RCT by re-computing $SUM(t[\hat{m}])$ for all the rows in the RCT that have the *next*-th bit of the BA set to one (line 17). Overall, we only access D twice in total (rather than twice per loop): once to compute the RCT and once to write out the updated $t[\hat{m}]$s after iterative scaling has converged (lines 19-21). Since the RCT is likely to be much smaller than D and can be replicated on each node, the runtime of iterative scaling can decrease significantly (as we will show in Section VI).

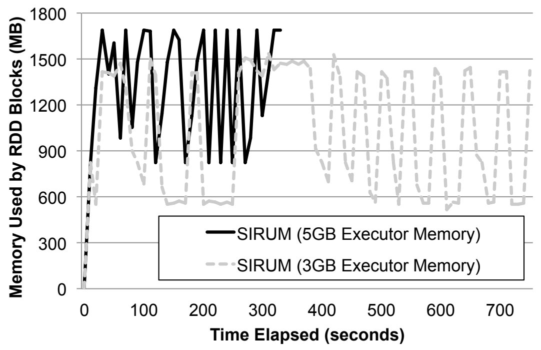

Fig. 7: Memory usage over time: different memory allocations

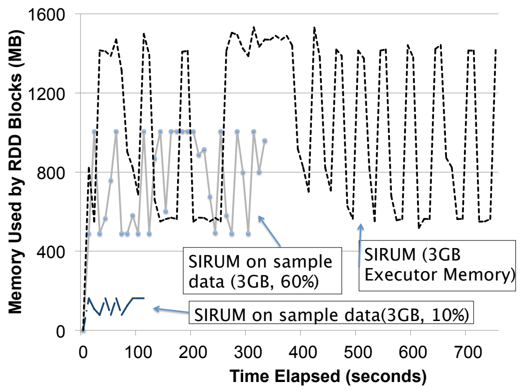

Fig. 8: Memory usage over time: SIRUM vs. SIRUM on sample data

B. Dealing with Very Large Datasets

Even with the above optimizations, we cannot completely avoid accessing D. If D does not fit in the main memory of the cluster, the performance penalty of HDFS I/O may be significant. In Spark, this problem is further magnified by the fact that Java objects often occupy more space in memory than data stored on disk.

To illustrate this problem, we run SIRUM with `Income` as the input dataset and $k = 10$ with two different amounts of available memory: 3GB and 5GB. We use a single node in this experiment; the findings are similar with more nodes. By default, around 60 percent of the allocated memory can be used to store the RDDs while the rest is mostly used for object creation. This gives roughly 1.8 and 3GB of RAM, respectively, for RDDs.

Figure 7 plots the memory used by RDDs as a function of elapsed time. With 5GB of memory, SIRUM is twice as fast and uses more memory for RDDs. The poor performance with only 3GB of memory is due to insufficient memory to cache the entire data set, causing continuous data read from HDFS. With 5GB of memory, SIRUM stops reading data from HDFS after the input data is fully loaded in memory.

Motivated by the need to avoid disk I/O, we implement a strategy that we call *SIRUM on sample data* for very large datasets, in which we draw a random sample of D of size equal to the available memory, and use it instead of D during rule generation and iterative scaling. In Figure 8, we plot the memory usage over time, with 3GB of memory for SIRUM and SIRUM on sample data with 60 percent and 10 percent sampling rates. With either sampling rate, the data fit in memory (no disk I/O) and runtime decreases significantly, especially with 10 percent sampling. The downside of sampling is that the KL Divergence of SIRUM on sample data may be larger than that of SIRUM since the former is not using all of D to compute information gain and iterative scaling. We will experimentally evaluate this issue in Section VI-F.

VI. Experiments

This section presents our experimental results. We start by describing the experimental environment in Section VI-A. Next, we justify the use of Spark as a data processing platform by comparing the performance of Baseline SIRUM on Spark, Hive, SparkSQL and PostgreSQL (Section VI-B). In Section VI-C, we evaluate the information gain of Column Partitioning (CP) compared to Baseline SIRUM and in Section VI-D, we measure the performance of CP on wide datasets. In Section VI-E, we isolate the performance improvement of our optimizations of iterative scaling geared towards tall datasets. Finally, we study the scalability of SIRUM on sample data in Section VI-F.

In this paper, we do not demonstrate the advantage of SIRUM's maximum-entropy approach compared to other possible ways to generate informative rules such as decision trees. This has already been shown in previous work [1].

A. Setup

We implemented SIRUM using Spark 1.4.0, with YARN as the resource manager, and we also implemented Baseline SIRUM using SparkSQL 1.4.0, PostgreSQL 9.4.4, and Hive 1.2.0/Hadoop 2.6.0. We used a dedicated cluster of 16 nodes, each running CentOS 6.4 with 4 Intel Xeon E5-2620 2.10 GHz 6-core CPUs and 64 GB of DDR3 RAM. Unless stated otherwise, we used the following Spark parameters:

- Number of Spark executors = 16, one per node (plus one additional thread for the driver program).
- Memory allocated to each executor = 45 GB (leaving 19 GB per node for the operating system, Spark overhead, the driver program, etc.).
- Number of RDD partitions = 384, one for each CPU in the cluster. Each partition is assigned to a Spark task.

For iterative scaling, we used $\epsilon = 0.01$.

We repeated each experiment five times, dropped the highest and lowest runtimes, and took the average of the remaining three. In some experiments, we also measured the information gain of a set of rules, which we define as the KL-Divergence using just the all-stars rule minus the KL-Divergence using the given set of rules (lower KL-Divergence is better).

We used the following data sets, stored as CSV files in HDFS.

- `Income` contains U.S. Census data with household demographic attributes such as the number of children and marital status, which we used as dimension attributes. This dataset also includes household income. We converted income into a binary measure attribute indicating whether the given household's income exceeds $\$100,000$. This dataset was downloaded from IPUMS-USA at https://usa.ipums.org/usa/data.shtml/ and contains roughly 1.5 million tuples, 9 dimension attributes and 78 million possible rules. The dataset occupies 50 MB in HDFS.

- `SUSY` is generated using Monte Carlo simulations for distinguishing between a signal that produces supersymmetric particles and a background process which does not. The data set was found at https://archive.ics.uci.edu/ml/datasets/SUSY and contains roughly 5 million tuples, 18 dimension attributes and 68 billion possible rules. We convert the attribute values from real numbers to discrete values through bucketing, with three buckets per attribute. The dataset occupies 223 MB in HDFS.
- `GDELT` is an extract from the GDELT Event Database, which records over 300 categories of events around the world such as riots and diplomatic exchanges, with nearly 60 attributes including location and who is involved. The measure attribute is the number of mentions of an event. This data set was downloaded from gdeltproject.org/data.html and contains roughly 3.8 million tuples, 9 dimension attributes and 12 billion possible rules. The dataset occupies 141 MB in HDFS.
- `TLC` is provided by the New York City Taxi and Limousine Commission (TLC), and includes all trips completed by yellow taxis from 2009 to 2014. We selected 9 dimension attributes including the month of year when the trip is recorded, the number of passengers, payment method, longitude/latitude of pickup/dropoff locations, etc., and 1 measure attribute, the total payment. This dataset contains 1.08 billion rows and is available at www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. The dataset occupies 8.5 GB in HDFS.
- `ONP` (Online News Popularity) describes Mashable articles over a duration of two years with a heterogeneous set of features, which are normalized to categorical dimension attributes for our experiments. The measure attribute is the number of shares of the article in social networks. The data set was found at https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity. It has 39797 tuples and 58 dimension attributes including features such as the number of words in the title. The dataset occupies 4.8 MB in HDFS.
- `GasSensor` is a collection of measurements from chemical sensors. The data set was downloaded from https://archive.ics.uci.edu/ml/datasets/Gas+Sensor+Array+Drift+Dataset+at+Different+Concentrations. It contains 13910 tuples and 129 dimension attributes. The measure attribute is the concentration level of gas. The dataset occupies 3.6 MB in HDFS.

These datasets have different characteristics and allow us to test various aspects of our solutions. For example, `Income` is relatively short and narrow, while `GDELT` and `TLC` are progressively taller while still being narrow. On the other hand, `SUSY` is wide but can still be handled by Baseline SIRUM, whereas `ONP` and `GasSensor` are even wider and can only be handled by our optimized SIRUM.

Additionally, we create a dataset by drawing a random sample of 160 million rows from the full `TLC` dataset (denoted by *TLC_160m*). This is the largest fraction of `TLC` that fits within the allocated memory of the cluster. Note that `TLC` only takes 8.5gb in HDFS but occupies much more RAM that that after de-serialization. Additionally, only part of the memory is allocated to storing the input data while the rest is reserved for the java heap (memory allocated for computation), OS/YARN, etc.

B. Choice of Data Processing Platform

We start by comparing Baseline SIRUM[2] on Spark, SparkSQL, Hive and PostgreSQL to justify our use of Spark as the underlying platform. We show representative results using $k = 10$ and $|s| = 16$ for sample-based candidate pruning; other datasets and parameter choices lead to similar conclusions. Figure 9 compares the runtime of Baseline SIRUM on Spark and PostgreSQL using a single compute node with `Income`, whereas Figure 10 compares the runtime of Baseline SIRUM on Spark and Hive using the entire cluster with *TLC_160m*. In the former, PostgreSQL is six times slower likely because it is single-threaded and does not leverage multiple cores as Spark does (and obviously because it runs on a single node). In the latter, Hive is an order of magnitude slower. We found that the major bottleneck for SIRUM on Hive is the disk and network I/O for saving and retrieving intermediate results. In contrast, Spark caches parts of, if not all, intermediate results as RDDs. We also observed that launching and cleaning up tasks are slower for SIRUM on Hive. While *TLC_160m* fits in the main memory of the cluster, we will separately evaluate the performance of SIRUM on the full `TLC` dataset in Section VI-F.

We also implemented Baseline SIRUM using SparkSQL. The performance was worse than our hand-optimized implementation using Spark data operators. Through further analysis, we observed that SparkSQL translated queries into different execution plans which were less efficient than their counterparts using Spark data operators. Hence, we chose Spark data operators in the following sections. It is part of our future work to evaluate SIRUM on newer versions of SparkSQL.

C. Baseline SIRUM vs. SIRUM with Column Partitioning

Recall that Column Partitioning (CP) considers a smaller set of candidate rules than Baseline SIRUM. In this section, we investigate the drop in information gain of the resulting rules due to CP. Since Baseline SIRUM is intractable for wide datasets such as `ONP`, we can only compare its information gain to CP's information gain on narrower datasets, the widest one of these being `GDELT`. Recall that CP has two parameters: d_0, the number of columns per group, and p, the number of random partitioning schemes. We use the syntax CP_d_0_p; e.g., CP_16_10 correspond to ten partitioning schemes in which every group has at most 16 columns.

Figure 11 compares the information gain of CP against that of Baseline SIRUM as a function of k, the number of rules to be generated. We use the `GDELT` dataset which has

[2]including sample-based candidate pruning and broadcast joins, as described in Section III-A.

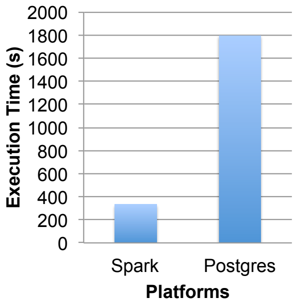

Fig. 9: Baseline SIRUM on Spark vs. PostgreSQL (`Income`)

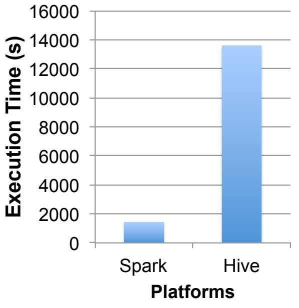

Fig. 10: Baseline SIRUM on Spark vs. Hive (*TLC_160m*).

.

9 dimension attributes and we test three CP variants: CP_3_3 (Three groups of three columns, partitioning repeated 3 times, giving a total of $3\times3=9$ groups of 3 columns), CP_2_3 (Five groups of up to 2 columns, partitioning repeated 3 times, giving a total of $3\times5=15$ groups of 1-2 columns) and CP_2_1 (Five groups of up to 2 columns, partitioning done only once, giving a total of five groups of 1-2 columns).

From Figure 11, we conclude that CP_3_3 achieves almost the same information gain as Baseline SIRUM. Furthermore, decreasing d_0 (CP_2_3) led to a slight drop in information gain, especially for small values of k. Decreasing p (CP_2_3 vs. CP_2_1) did not have a noticeable effect.

Figure 12 compares CP against Baseline SIRUM on the `SUSY` dataset which has 18 dimension attributes. With 6 dimension attributes in each group and 3 random partitioning schemes, CP_6_3 achieves almost the same information gain as Baseline SIRUM. Upon further inspection, we found that CP found some of the same rules as Baseline, and some different ones; the different rules had fewer constants than those found by Baseline but nearly the same information gain. This suggests why CP performs well: even if it misses the best rules as identified by Baseline, it finds other rules that have fewer constants and have nearly as much information gain. As for the impact of d_0 and p, as expected, CP_6_1 and CP_3_1 achieve lower information gain than CP_6_3 but the difference is within 20%.

Our results show that *CP is able to achieve information gain comparable to that of Baseline SIRUM with a small number of partitioning schemes and few attributes per group*. We only showed results for the first 10 rules because the marginal increase in information gain drops as k increases and therefore the information gain curves and the differences between CP and Baseline do not change much beyond $k=10$.

D. SIRUM on Wide Datasets: Performance of Column Partitioning

We now evaluate CP on the widest datasets we have: `ONP` (58 dimension attributes) and `GasSensor` (129 dimension attributes). Baseline SIRUM is intractable for such wide datasets, so we can only evaluate CP with various parameter choices. Figure 13 shows the information gain of CP over the `ONP` dataset with k, the number of rules, on the x-axis. Each of the

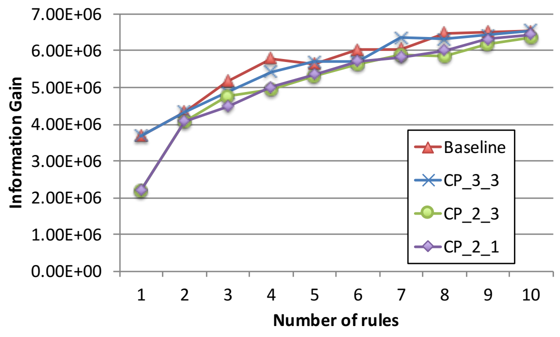

Fig. 11: Comparing information gain of CP and Baseline (`GDELT`, $k=10$, $|s|=64$)

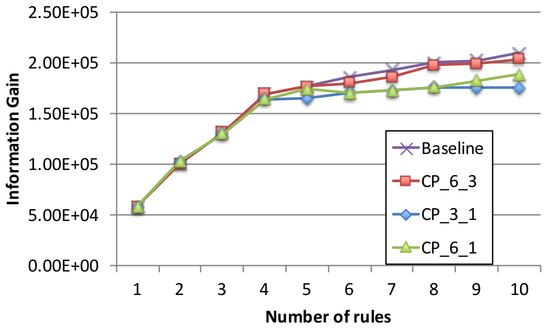

Fig. 12: Comparing information gain of CP and Baseline (`SUSY`, $k=10$, $|s|=64$)

four tested CP variants has $d_0=18$, meaning that there are $\lceil 58/18 \rceil=4$ groups of up to 18 columns. We vary p from 1 till 16. Interestingly, when comparing CP_18_1 with CP_18_2, there is only a minor increase in information gain between rule 5 and rule 18. Furthermore, there is not much information gain improvement compared to CP_18_4 and CP_18_16. In particular, as k approaches 30, there is not much difference even between CP_18_1 and CP_18_16.

Figure 14 shows the corresponding runtimes for $k=30$. These results are not surprising since, for example, CP_18_16 considers 16 random partitionings, which is 16 times the work of CP_18_1. From these results, we conclude that *a relatively small value of p provides a good tradeoff between efficiency and information gain for CP*.

Figure 15 and 16 show the results of the same experiment on the `GasSensor` data set. Here, $\lceil 129/18 \rceil=8$ groups of up to 18 columns and we test p from 1 to 14. As before, there is not much improvement in information gain as p increases, but of course, the runtime increases. Again, we conclude that a small value of p works well.

E. SIRUM on Tall Datasets: Improved Iterative Scaling

The next set of experiments isolates the performance improvement of RCT from Section V compared to Baseline SIRUM. Here, we only measure the runtime of iterative scaling and omit the rule generation time which is not affected. Figures 17 and 18 show the total runime of iterative scaling

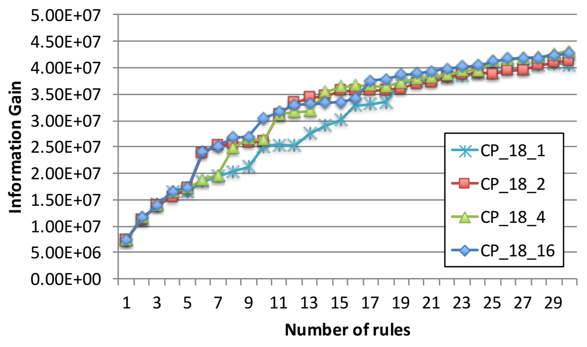

Fig. 13: Comparing the information gain of CP as we increase the number of partition schemes p (ONP, $k = 30$, $|s| = 64$)

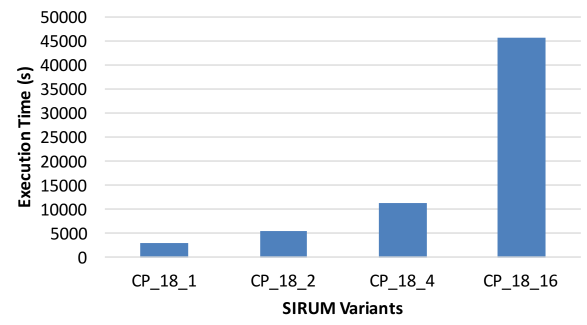

Fig. 14: Comparing the rule generation execution time of CP as we increase the number of partition schemes p (ONP, $k = 30$, $|s| = 64$)

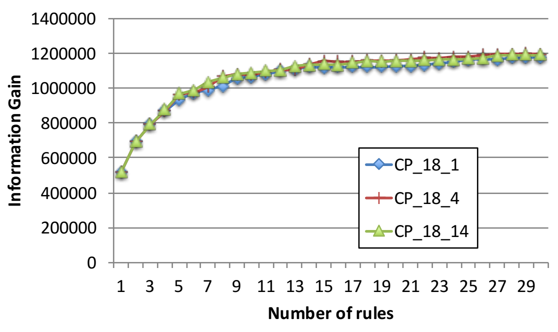

Fig. 15: Comparing the information gain of CP as we increase the number of partition schemes p (GasSensor, $k = 30$, $|s| = 64$)

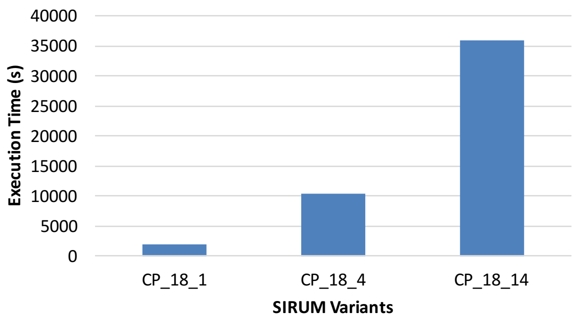

Fig. 16: Comparing the rule generation execution time of CP as we increase the number of partition schemes p (GasSensor, $k = 30$, $|s| = 64$)

alone for GDELT and SUSY, respectively, for different values of k, the number of rules. We conclude that *SIRUM with RCT performs iterative scaling four to five times faster on both datasets and for all tested values of* k.

F. Scaling Beyond Main Memory: SIRUM on Sample Data

Finally, we examine the performance of SIRUM on sample data proposed in Section V-B. Recall that the idea is to sample the dataset so that it fits in memory and run SIRUM as though the sample were the full dataset. We now show that *SIRUM on sample data is significantly faster and can scale to larger data sets, while the KL-divergence penalty from working with a sample is small.*

Figures 19 and 20 illustrate the running time and information gain as we vary the sampling rate from 100 percent down to ten, one and 0.1 percent. The former uses TLC and 45 GB of memory for each of the 16 executors; the latter uses SUSY and 3 GB executor memory on a single node; both use $|s|$=16. We ensure that in both cases, the dataset is larger than the available executor memory. Note that the x-axis is logarithmic.

In both cases, there is already a significant performance improvement with a ten percent sample, which is small enough to be cached in memory: for TLC, the runtime decreases by an order of magnitude and for SUSY, it decreases by a factor of about 4. At the same time, the drop in information gain is very small. In fact, even a one-percent sampling rate does not decrease the information gain very significantly, but running time decreases further still. Eventually, though, information gain suffers and running time no longer decreases, so there is a limit to how much we can sample the data for the purpose of rule generation and iterative scaling. In particular, for the tested datasets, it appears that one percent is the lowest reasonable sampling rate. Notably, even a one-percent sample of TLC contains over 10 million rows and a one-percent sample of SUSY contains roughly 50 thousand rows, which is enough to generate good rules and is of course far larger than the very small samples used in sample-based candidate pruning.

VII. Related Work

Informative rule mining has appeared in several contexts, including data summarization (e.g., of multi-dimensional data with a binary measure attribute [1] and itemset data [2]) and data cube exploration [3]. Diagnosing data quality problems can also benefit from informative rule mining, although previous work in this area such as Data X-Ray [4] and Data Auditor [5] used different techniques. SIRUM may be used in all of these applications to improve performance and scalability.

Some of the performance improvements proposed in existing work optimize specific sub-problems; e.g., finding informative itemsets [2]. Others limit the space of candidate rules; e.g., by disallowing overlapping rules unless one is contained within the other [3], or by only considering those which

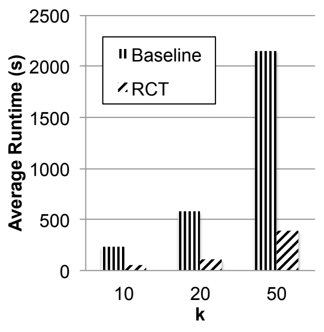

Fig. 17: Performance improvement of RCT (GDELT)

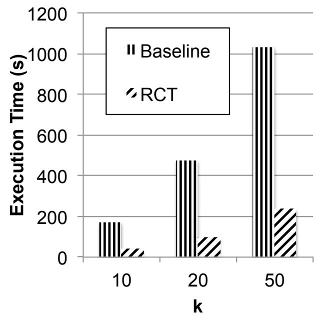

Fig. 18: Performance improvement of RCT (SUSY)

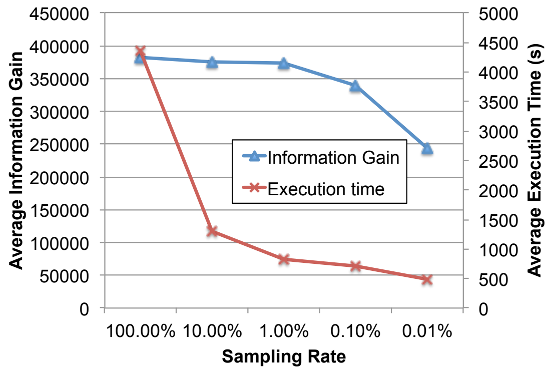

Fig. 20: Execution time and information gain of SIRUM on sample data over the SUSY dataset (8 GB executor memory)

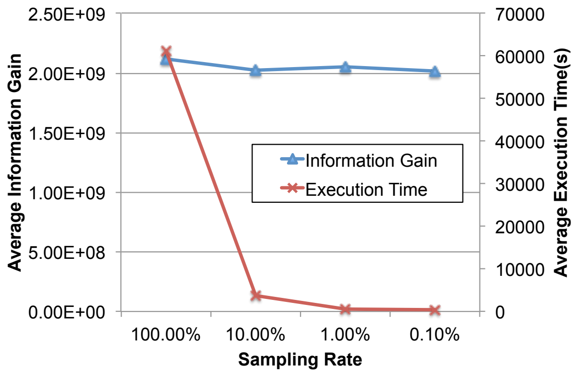

Fig. 19: Execution time and information gain of SIRUM on sample data over the TLC dataset (16 X 45 GB executor memory)

can be generated from a random sample, i.e., sample-based candidate pruning [1]. SIRUM includes new optimizations of iterative scaling and new candidate rule pruning techniques (optimizations of iterative scaling have been proposed in [2], but in the context of itemsets; our optimizations are more general). Furthermore, we know of no other work on distributed informative rule mining. Finally, SIRUM on sample data (recall Section V-B) is akin to other work on sample-based computation over big data such as BlinkDB [14].

VIII. Conclusions

In this paper we presented SIRUM, a distributed approach to informative rule mining that scales to wide and tall datasets. We proposed several optimizations of the rule mining process, including a new technique for iterative scaling that optimizes away table scans, and a new method that effectively prunes the search space of candidate rules. We implemented SIRUM in the Spark distributed processing system, and evaluated it on several real datasets. We showed that our optimizations improve performance and that we can handle much taller and wider tables than a distributed implementation of the previous state of the art.

We have three directions in mind for future work. First, we will develop a principled approach to estimate SIRUM parameters in real applications. Second, we want to build a streaming version of SIRUM (e.g., using Spark Streaming) that incrementally maintains informative rules as new data arrive. Third, we want to study the problem of informative rule generation to simultaneously provide information about multiple measure attributes.

References

[1] K. El Gebaly, P. Agrawal, L. Golab, F. Korn, and D. Srivastava, "Interpretable and informative explanations of outcomes," *Proc. of VLDB Endowment* 8(1): 61-72, 2014.

[2] M. Mampaey, N. Tatti, and J. Vreeken, "Tell me what I need to know: succinctly summarizing data with itemsets," *Proc. of the ACM SIGKDD Int. Conf. on Knowl. Discovery and Data Mining*, 2011, 573-581.

[3] S. Sarawagi, "User-cognizant multidimensional analysis," *The VLDB Journal*, 10(2): 224-239, 2001.

[4] X. Wang, X. L. Dong, and A. Meliou, "Data X-Ray: A diagnostic tool for data errors," *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2015, 1231-1245.

[5] L. Golab, H. Karloff, F. Korn, and D. Srivastava, "Data auditor: Exploring data quality and semantics using pattern tableaux," *Proc. of VLDB Endowment*, 3(1), 1641-1644, 2010.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *Proc. of the USENIX Conf. on Networked Systems Design and Implementation*, 2012, 2.

[7] A. Berger, "The improved iterative scaling algorithm: A gentle introduction," 1997.

[8] J. N. Darroch and D. Ratcliff, "Generalized iterative scaling for log-linear models," *The Annals of Mathematical Statistics*, 1470-1480, 1972.

[9] I. Csiszár, "I-divergence geometry of probability distributions and minimization problems," *The Annals of Probability*, 146-158, 1975.

[10] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan, "Data cube materialization and mining over mapreduce," *Transactions on Knowledge and Data Engineering* 24(10): 1747-1759, 2012.

[11] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce," *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2010, 975-986.

[12] S. Chen, "Cheetah: a high performance, custom data warehouse on top of mapreduce," *Proc. of VLDB Endowment* 3(1): 1459-1468, 2010.

[13] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates," *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1996, 506-521.

[14] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," *Proc. of the ACM European Conf. on Computer Systems*, 2013, 29-42.