Scalable Informative Rule Mining

by

Guoyao Feng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis 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 concise set of rules (patterns) over the values of the dimension attributes that provide the most information about the distribution of a numeric measure attribute. SIRUM optimizes this task for big, wide and distributed datasets. We implemented SIRUM in Spark and observed significant performance improvements on real data due to our optimizations.

Acknowledgements

First, I would like to give the most sincere thanks to my supervisor, Professor Lukasz Golab, for his invaluable support and guidance, his patience and encouragement. This thesis would not have been possible without his help and dedication. My sincere thanks also go to Dr. Divesh Srivastava for his insights and feedback.

I would also like to thank my committee members, Professor Tamer Özsu, Professor Khuzaima Daudjee and Professor Srinivasan Keshav, for the time and energy they devoted to reading this thesis and to providing comments and feedback.

Dedication

This is dedicated to my parents for their endless love, support and encouragement.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we present the SIRUM system for Scalable Informative RUle Mining from big, 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. We start with three sample applications of SIRUM as motivation.

Data Profiling and Summarization: For example, consider the multidimensional dataset as presented in Table 1.1. Each row in the dataset represents a flight identified by its flight ID. The other attributes record the amount of flight delay along with relevant information including the day of flight (Day), the cities of departure and arrival (Origin and Destination). Suppose that a data analyst plans to summarize the distribution of flight delay as a function of the different value combinations of the other attributes, possibly in preparation for developing a prediction model. She can treat the flight delay as the measure attribute and other attributes, except for the flight ID, as the dimension attributes. We will discuss the columns labelled $\hat{m}_i$ later.

Table 1.2 presents an example of informative rule set over the flight dataset. The rule set contains 4 rules total, each identified by its rule ID. Note that a wild-card symbol ('$*$') matches all possible values of the attribute. A rule is also augmented with the following aggregate values: (1) the average of measure attributes, AVG(Late), and (2) the number of tuples covered by the rule, COUNT(*). For instance, the first rule states that the 14 flights in the flight dataset were late by 10.4 minutes on average. The second rule states that the 4 flights arriving in London were late by 15.3 minutes on average.

The informative rule set can be built incrementally. After the first rule covering all tuples is added to the rule set, adding the second rule provides the greatest amount of additional information content about the dataset. The intuition is twofold. First, there are a sufficiently large number

Table 1.1: A flight delay report

Flight ID	Day	Origin	Destination	Delay	$\hat{m}_1$	$\hat{m}_2$	$\hat{m}_3$
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

of flights arriving in London. Second, the average delay for London-bound flights exhibits the greatest deviation from the average delay of all flights. Following the same intuition, the third rule is considered to have the greatest additional information content based on the distribution of AVG(Late) entailed by the first two rules. We will formalize the notion of information content in the following sections.

Table 1.2: An example of informative rule set over the flight dataset

Rule ID	Day	Origin	Destination	AVG(Late)	count
1	*	*	*	10.4	14
2	*	*	London	15.3	4
3	Fri	*	*	18	2
4	Sat	*	*	16	2

Smart Exploration of Data Cubes: Informative rule mining helps guide the analyst explore the data cube if the analyst possesses prior knowledge about the dataset such as a predefined

set of interesting cells [29]. 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 1.3. SIRUM recommends the two rules shown at the bottom of Table 1.3, 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.

Table 1.3: Two tuples corresponding to a user's prior knowledge, followed by two informative rules, over the flight dataset

Day	Origin	Destination	AVG(Late)	count
*	*	*	10.4	14
*	SF	*	11.8	4
*	*	London	15.3	4
Fri	London	LA	16	1

Data Cleansing: Informative rule mining is useful in diagnosing data quality issues (see, e.g., Data X-Ray [35] and Data Auditor [17]). In this problem, the measure attribute denotes the quality of a tuple (e.g., 1=dirty and 0=clean) 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. For instance, Table 1.4 presents a sample with 8 dimension attributes from the `GDELT` data set[1]. In each row, the dimension attributes record the properties of a global event and the measure attribute outputs whether the type of the second actor(s)[2] of the event is missing. From the sample dataset, SIRUM is able to produce the rules listed in Table 1.5 to highlight the 'dirty' records. It shows that events whose attribute values match those in the second rule have an average of 1 in the measure value, which is significantly higher than the overall average (0.33). An average value of 1 indicates that none of these events record the type of the second actor(s).

Informative rule mining is *interactive*: a user may request an initial list of informative rules,

[1] `GDELT` is a real data set we will use to evaluate SIRUM in Chapter 5. It is published by a project that monitors the world's news media in print, broadcast and web formats [23]. It documents global events on a daily basis.

[2] Normally a GDELT event is recorded in an expanded version of the dynamic CAMEO format [31], capturing two actors and the action performed by Actor1 upon Actor2.

[3] The CAMEO event codes are defined in a taxonomy of three levels. For events at the third level, the event base code keeps track of its level two leaf node. For example, code "0251" (Appeal for easing of administrative sanctions) would return an Event Base Code of "025" (Appeal to yield). It allows the project to aggregate events at various resolutions of granularity. For events at the second or first level, this is the same as the Event Code [23].

Table 1.4: A sample from GDELT data set

Event ID	Actor1 Country	Actor1 Type	Is Root Event	Event Base Code[3]	Event Class	Actor1 Geo-Type	Actor2 Geo-Type	Action Geo-Type	Is Actor2 Type Missing
1	US	Media	1	112	Verbal Conflict	US STATE	US CITY	US CITY	0
2	US	Media	1	173	Material Conflict	US CITY	US CITY	US CITY	1
3	US	Media	1	173	Material Conflict	US CITY	US STATE	US CITY	1
4	US	Political Opposition	0	114	Verbal Conflict	US STATE	US CITY	US CITY	0
5	US	Rebels	0	36	Verbal Cooperation	US CITY	US CITY	US CITY	0
6	US	NGO	0	51	Verbal Cooperation	WORLD CITY	WORLD STATE	WORLD CITY	0

explore the data, and ask for more rules (for the same or a different measure attribute). This motivates the need for efficiency. However, challenges arise when computing informative rules over big data. Nowadays, it is common to collect very large volumes of data, both in terms of the number of entities and in terms of the number of descriptive attributes about a given entity. For example, a flight data set may contain millions of flights, with many dimension attributes corresponding to the wide 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/dimension attributes), leading to a very large number of candidate rules.

Big data are typically processed using distributed computing platforms. Here, efficiency

Table 1.5: An example of informative rules to highlight dimension attribute values correlated to records without Actor2 Type

Actor1 Country	Actor1 Type	Is Root Event	Event Base Code	Event Class	Actor1 Geo-Type	Actor2 Geo-Type	Action Geo-Type	AVG (Late)	count
*	*	*	*	*	*	*	*	0.33	6
US	Media	1	*	Material Conflict	US CITY	*	US CITY	1	2
US	*	*	173	Material Conflict	US CITY	US CITY	*	1	1

matters not only to enable interactive applications, but also to save money if the computation happens on the cloud (e.g., using Amazon EC2 or Microsoft Azure) and costs are incurred based on resource utilization. Unfortunately, distributed generation of informative rules is challenging for several reasons. It is an iterative process, which repeatedly selects the next most informative rule (details to follow in Chapter 2). 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.

1.1 Contributions

Despite the recent interest in informative rule mining [16, 24, 29], existing techniques do not scale to tall and wide datasets: they require multiple scans of large datasets and do not leverage parallel computation. The evaluation of these techniques has also been limited to short and narrow datasets on centralized systems. 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 on top of Spark [37], 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 Chapter 5 by comparing it to versions of SIRUM implemented using a traditional MapReduce-based framework (Hive), SparkSQL and PostgreSQL.

2. A profiling study that reveals the bottlenecks of distributed informative rule generation from tall and wide tables.

3. Optimizations for distributed informative rule generation from tall and wide tables, focusing on reducing data shuffling, CPU and memory usage, and disk I/O.

4. Experimental evaluation of SIRUM compared to prior work [16, 29], showing up to an order of magnitude performance improvements of rule mining and data cube exploration on real datasets.

1.2 Outline

The rest of the thesis is structured as follows. Chapter 2 presents background information on informative rule mining. Chapter 3 outlines a baseline implementation of SIRUM and profiles its performance on real datasets. Chapter 4 presents various performance optimizations over the baseline implementation of SIRUM. Chapter 5 presents our experimental results. Chapter 6 discusses previous work. Chapter 7 concludes the thesis.

Chapter 2

Preliminaries

2.1 Definitions and Notations

Consider a relational data set D with d dimensional attributes, $\{A_1, A_2, \ldots, A_d\}$, and a numeric measure attribute, m. A tuple from D is denoted by t. The output is a list of k informative rules R. The value of k is supposed to be small such that the rule list is interpretable by human beings. A rule, denoted by r, is essentially a tuple from a multidimensional space, $(dom(A_1) \cup \{*\}) \times \cdots \times (dom(A_d) \cup \{*\})$. We use $t \asymp r$ to denote the fact that t matches r, or equivalently r covers t. t matches r if and only if either $r[A_j] =$ '*' or $t[A_j] = r[A_j]$ for each dimension attribute A_j. For instance, the tuple t_6 in Table 1.1 matches rules r_1, r_2 and r_4 in Table 1.2, but not r_3.

We define the *support set* of a rule r, $S_D(r)$, to be the set of tuples covered by r, that is, $S_D(r) = \{t | t \asymp r, t \in D\}$. For example, the support set of r_3 in Table 1.2 contains t_1 and t_2 from Table 1.1. We use $m(r)$ to denote the average value of the measure attribute of tuples covered by r, i.e., $m(r) = \frac{1}{|S_D(r)|} \sum_{t \in S_D(r)} t[m]$. In Table 1.2, the columns AVG(Late) and count(*) show $m(r)$ and $|S_D(r)|$ for each rule respectively.

Given a tuple t, we denote its estimated value of $t[m]$ by $t[\hat{m}]$. Similarly, the average estimated value of the measure attribute of tuples covered by r is $\hat{m}(r) = \frac{1}{|S_D(r)|} \sum_{t \in S_D(r)} t[\hat{m}]$. We will describe how to determine $t[\hat{m}]$ based on the set of matching rules in Section 2.2.

Given two tuples t_i and t_j, we further define the *least common ancestor* (LCA) of t_i and t_j. Let $r = lca(t_i, t_j)$ be the least common ancestor. For every $A_l \in \{A_1, A_2, \cdots, A_d\}$, $r[A_l] =$

Table 2.1: Frequently used symbols in this paper

Symbol	Meaning
D	A dataset
t	A tuple
$t[A_j]$	The value of the A_j attribute of tuple t
R	The list of rules already generated
r	A rule
$S_D(r)$	The set of tuples in D covered by r
k	The number of rules to be generated
A_j	A dimension attribute of D
$dom(A_j)$	The active domain of A_j
d	The number of dimension attributes
m	A measure attribute of D
$\hat{m}$	An estimate of m
$m(r)$	Average value of $t[m]$ of the tuples matching r
$\hat{m}(r)$	Average value of $t[\hat{m}]$ of the tuples matching r

$t_i[A_k]$ if $t_i[A_l] = t_j[A_l]$; otherwise, $r[A_l] =$ "$*$". For example, consider t_1 and t_6 from Table 1.1. Their LCA is (*, *, London), which is exactly r_2 in Table 1.2.

A rule r_1 is *disjoint* from another rule r_2 if they satisfy the following conditions: there exists at least one A_i such that (1) $r_1[A_i] \neq *$; (2) $r_2[A_i] \neq *$; and (3) $r_1[A_i] \neq r_2[A_i]$. Equivalently, r_1 and r_2 are *overlapping* if the condition fails to hold. Note that the relation is solely based on their attribute values rather than their the support sets; two rules being disjoint implies that their support sets are disjoint, but the converse is not true. For example, consider the following pair of rules: (Fri, London, LA) and (*, SF, LA). Since they have different values in the 'Origin' attribute, they are by definition disjoint and their support sets ($\{t_2\}$ and $\{t_1, t_{11}\}$ respectively from Table 1.1) must also be disjoint. On the contrary, consider another pair of rules (Wed, *, *) and (*, *, London) and their corresponding support sets: $\{t_8\}$ and $\{t_1, t_4, t_6, t_{11}\}$ from Table 1.1. While their support sets are disjoint, the pair of rules is overlapping by definition because it is possible to define a tuple (Wed, Chicago, London) that matches both rules.

Table 2.1 lists the symbols frequently used in the remainder of this paper.

2.2 Maximum Entropy Principle

In Table 1.2, each rule is testable information, i.e., a statement about the average value of the measure attribute with respect to the value combinations of dimension attributes. The set of rules allows us to approximate the true distribution of the measure attribute. The estimated value of measure attribute can be computed following the *principle of maximum entropy*, which states that subject to testable information, the probability distribution which best represents the current state of knowledge is the one with the highest entropy [18]. In the case of informative rule mining, the maximum entropy approximation of $t[m]$ is the solution to the following optimization problem.

$$\begin{aligned} \text{minimize} \quad & \sum_{t \in D} -t[\hat{m}] \cdot \log(t[\hat{m}]) \\ \text{subject to} \quad & \sum_{t \in S_D(r)} t[m] = \sum_{t \in S_D(r)} t[\hat{m}] \quad \forall r \in R \\ & t[\hat{m}] \geq 0 \qquad\qquad\qquad\qquad \forall t \in D \\ & \sum_{t \in D} t[\hat{m}] = 1 \end{aligned} \tag{2.1}$$

The use of maximum entropy implies that $t[m]$ must satisfy the following conditions:

1. for all $t \in D$, $t[m] \geq 0$
2. $\sum_{t \in D} t[m] = 1$

If the measure attribute does not satisfy the conditions above, we can apply the following transformations and reduce it to the optimization problem 2.1.

1. Suppose there exists at least one $t[m]$ such that $t[m] < 0$. Since the number of tuples is finite, there must exists a value $M < 0$ such that $M \leq t[m]$ for all $t \in D$. Let $t[m'] = t[m] - M$. It follows that $t[m'] \geq 0$ for all $t \in D$.
2. If $\sum_{t \in D} t[m] = 0$, define $t[m'] = t[m] + \dfrac{1}{\|D\|}$ such that $\sum_{t \in D} t[m'] = 1$.
3. If $\sum_{t \in D} t[m] \neq 1$ or 0, define $t[m'] = \dfrac{t[m]}{\sum_{t \in D} t[m]}$ so that $\sum_{t \in D} t[m'] = 1$.

Once the solution for $t[m']$ is found, we will perform a reverse transformation to calculate $t[m]$.

In the following sections, we assume that $r_1 = (*, *, \cdots, *)$ is always the first rule selected. It allows us to extend the second condition to $\sum_{t \in D} t[m] = C$ where $C \neq 0$. Suppose that $t[m]$ satisfies the new condition, i.e., $\sum_{t \in D} t[m] = C \neq 0$. First, we normalize $t[m]$ to $t[m'] = \frac{t[m]}{C}$. It follows that $\sum_{t \in D} t[m'] = 1$. Now we define a relaxed optimization problem from optimization problem 2.1 by removing its last constraint, $\sum_{t \in D} t[\hat{m}] = 1$. Let $t[\hat{m}']$ be the solution to the relaxed problem for $t[m']$. Now we show that $t[\hat{m}']$ also satisfies the constraint that $\sum_{t \in D} t[\hat{m}'] = 1$. Since the first rule r_1 covers all tuples, we have

$$\sum_{t \in D} t[\hat{m}'] = \sum_{t \in S_D(r_1)} t[\hat{m}'] = \sum_{t \in S_D(r_1)} t[m'] = \sum_{t \in D} t[\hat{m}'] = 1$$

. It means $t[\hat{m}']$ is indeed a solution to the optimization problem 2.1. It follows that $t[\hat{m}] = C \cdot t[\hat{m}']$ is an approximation of $t[m]$ based on the maximum entropy principle. Thus, if $\sum_{t \in D} t[m] = C \neq 0$, selecting $r_1 = (*, *, \cdots, *)$ as the first rule allows us to find a solution for $t[m]$ using the maximum entropy principle.

Going back to the example in Table 1.1 and 1.2, the column labelled $\hat{m}_1$ shows the estimated values of flight delay based on the first rule, which covers all tuples in the entire data set. Given that the average flight delay is 10.4 in the entire dataset, the maximum-entropy solution sets each $t[\hat{m}]$ to 10.4. After adding r_2, t_1, t_4, t_6 and t_{11} must be assigned $t[\hat{m}] = 15.3$ to satisfy $\sum_{t \in S_D(r_2)} t[m] = \sum_{t \in S_D(r_2)} t[\hat{m}]$, or $m(r_2) = \hat{m}(r_2)$. Moreover, the maximum-entropy solution has to set the other $t[\hat{m}]$'s to 8.4 each to satisfy $m(r_1) = \hat{m}(r_1) = 10.4$, as shown in column $\hat{m}_2$. Following the same strategy, the column $\hat{m}_3$ shows the maximum-entropy solution for the first three rules.

Iterative Scaling

The optimization problem 2.1 can be solved using iterative scaling [7, 13]. For each tuple t, $t[\hat{m}]$ can be expressed as $\prod_{r, t \succeq r} \lambda(r)$, where $\lambda(r)$ is a *multiplier* associated with a rule r [12]. Algorithm 1 shows the procedure of iterative scaling. In SIRUM, the procedure is executed whenever a new rule is appended to the rule list.

At the beginning when the rule list is empty, the estimated value of measure attribute is set to 1 by default. As a rule r is first added to the list, $\lambda(r)$ is initially set to 1 by default and $m(r)$ is computed. The optimization problem imposes the constraint that $m(r) = \hat{m}(r)$. However, it is not uncommon to replace the constraint with $\frac{|m(r) - \hat{m}(r)|}{|m(r)|} < \epsilon$ where ϵ is a small positive number provided by the user (say, 0.01).

Algorithm 1 Iterative Scaling

Input: $D, R, \lambda, m(r)$ for all $r \in R$, ϵ
Output: $\hat{m}(r)$ for all $r \in R$

1: $DIFF \leftarrow ARRAY(|R|, 0)$
2: **while** true **do**
3: **for** $i \leftarrow 1$ 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: $DIFF[i] \leftarrow \frac{|m(r_i) - \hat{m}(r_i)|}{|m(r_i)|}$
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** $t \asymp r_{next}$ **do**
11: $t[\hat{m}] \leftarrow \prod_{r, t \asymp r} \lambda(r)$
12: **end for**
13: **else**
14: **break**
15: **end if**
16: **end while**

At this point, Algorithm 1 is ready to run as all inputs are in place. Line 1 of the algorithm assigns zeros to all elements of the DIFF array, which maintains the difference between $m(r)$ and $\hat{m}(t)$. In lines 3 through 6, it computes the current $\hat{m}(r)$ values for each rule, as well as their differences from the actual $m(r)$s. In line 7, it finds the rule with the greatest DIFF. If DIFF is greater than the threshold ϵ, it scales the multiplier for this rule as shown in line 9. Next, it must update the estimated values of measure attribute for all tuples matching this rule (line 11). The loop keeps iterating until all the rules have DIFFs below ϵ which means the $\hat{m}(r)$s have converged to $m(r)$s.

Now we illustrate the procedure of iterative scaling using the running example.

1. After r_1 is appended to the rule list, both $\lambda(r_1)$ and $t[\hat{m}]$ for all $t \in D$ are set to 1 initially. Thus $\hat{m}(r_1) = 1$. Since $m(r_1) = 10.4$, Algorithm 1 sets $\lambda(r_1) = 1 * \frac{10.4}{1} = 10.4$ (line 3

to 6) and updates $t[\hat{m}]$ to 10.4 for all $t \in D$ at line 11. At this point, $\frac{|m(r) - \hat{m}(r)|}{|m(r)|} < \epsilon$ holds and the algorithm exits the while loop at line 14 and terminates.

2. As r_2 is appended to the list, $\lambda(r_2)$ is set to 1 initially and the algorithm scales it through $\lambda(r_2) = \lambda(r_2) * \frac{m(r_2)}{\hat{m}(r_2)} = 1 * \frac{15.3}{10.4} = 1.47$. Consequently, $t[\hat{m}]$ for t_1, t_4, t_6, and t_{11} must be updated to $t[\hat{m}] = \lambda(r_1) * \lambda(r_2) = 15.25$ (line 11). At this point, $\hat{m}(r_1)$ has changed from 10.4 to 11.76 as a result of the updating $t[\hat{m}]$ and therefore now $m(r_1) \neq \hat{m}(r_1)$. So, we scale $\lambda(r_1)$ by setting it to $\lambda(r_1) * \frac{m(r_1)}{\hat{m}(r_1)} = 10.4 * \frac{10.4}{11.76} = 9.2$. 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 $\lambda(r_2) = 1.47 * \frac{15.25}{13.5} = 1.67$. 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 1.1.

2.3 Kullback-Leibler Divergence

In this section we quantify the quality of a rule set by measuring the difference between $t[m]$ and $t[\hat{m}]$. Kullback-Leibler divergence [21] (KL-divergence), also known as *information gain*, is a widely used information-based measure of disparity among probability distributions. Given two probability distributions P and Q defined over X, with Q absolutely continuous with respect to P, the KL-divergence from P to Q is defined as

$$D_{KL}(P||Q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

where p and q denote the density functions of P and Q.

In information theory, the *entropy* [32] of X, $H(X)$, can be interpreted as the expected code length for values of X according to optimal coding. The *cross-entropy* for Q on P, $H(P, Q)$, is the expected code length for values of X if a different distribution Q, rather than the true distribution P, is used to obtain optimal coding. We can interpret KL-divergence as a measure of the expected number of extra bits necessary if Q rather P is used to achieve optimal coding. That is,

$$D_{KL}(P||Q) = H(P, Q) - H(Q) = -\int p(x) \log q(x) dx - (-\int p(x) \log p(x) dx)$$

KL-divergence satisfies the following key properties:

1. Non-negativity: $D_{KL}(P||Q) \geq 0$ for all P, Q, with $D_{KL}(P||Q) = 0$ if and only if $t[m] = t[\hat{m}]$ for all $t \in D$ (based on Gibb's inequality [27])
2. Self-similarity: $D_{KL}(P||P) = 0$
3. Self-identification: $D_{KL}(P||Q) = 0$ only if $P = Q$

In this thesis, we use KL-divergence to measure the similarity between the distribution of $t[m]$ and the distribution of $t[\hat{m}]$. Intuitively, the KL-divergence has a positive value if the two distributions differ. As the estimation of $t[m]$ improves, the KL-divergence is expected to converge to zero, at which point $t[\hat{m}]$ has almost the same distribution as $t[m]$.

Return to the example in Table 1.1. By normalizing the values of $t[m]$ and $t[\hat{m}]$, KL-divergence can be used to measure the difference between their distributions. Specifically, the KL-divergence between $t[m]$ and $t[\hat{m}_1]$, after normalization, is

$$D_{KL}(t[m]||t[\hat{m}_1]) = \sum_{t \in D} t[m] \log \frac{t[m]}{t[\hat{m}_1]} = 4.1 \times 10^{-3}$$

. Similarly, for $t[\hat{m}_2]$ we have

$$D_{KL}(t[m]||t[\hat{m}_2]) = \sum_{t \in D} t[m] \log \frac{t[m]}{t[\hat{m}_2]} = 1.4 \times 10^{-3}$$

. By adding the second rule into the rule set, SIRUM reduces the KL-divergence from 4.1×10^{-3} to 1.4×10^{-3}.

2.4 Informative-based Rule Mining with Binary Measure Attributes

The information-based rule mining problem with binary measure attribute on a centralized data processing system has been studied in [16]. In particular, El Gebaly *et al.* prove that solving the following problem is NP-hard in the size of D.

Problem 1. Given a threshold τ and a multidimensional data set D with a binary attribute m, construct an informative rule list of the smallest size, with $t[\hat{m}]$ determined by the maximum entropy principle defined in Formulation 2.1, such that $D_{KL}(t[m]||t[\hat{m}]) < \tau$.

A greedy heuristic [16] is to incrementally construct the rule list, adding one rule at a time. At each step, the heuristic chooses the rule with the greatest reduction in KL-divergence, followed by iterative scaling. Given the current rule set R, however, computing the KL-divergence for a candidate rule r requires running iterative scaling over $R \cup \{r\}$. Since the number of candidate rules is very large, this approach is prohibitively expensive. Instead, we use an estimate of the KL-divergence without the need to run iterative scaling as if the rule is selected. We consider the difference between the sums of actual and estimated m values of only those tuples that match the rule. We denote the estimate as the information gain of r, which is defined below:

$$gain(r) = \sum_{t \in D, t \asymp r} t[m] \cdot \log \frac{\sum_{t \in D, t \asymp r} t[m]}{\sum_{t \in D, t \asymp r} t[\hat{m}]} \tag{2.2}$$

After a rule r is added to R, the first constraint in optimization problem 2.1 states that $\sum_{t \in S_D(r)} t[m] = \sum_{t \in S_D(r)} t[\hat{m}]$, which means its gain $gain(r) = \sum_{t \in D, t \asymp r} t[m] \cdot 0 = 0$. Note that any rule r_ω whose average m value of its support set is underestimated ($\sum_{t \in D, t \asymp r_\omega} t[m] > \sum_{t \in D, t \asymp r_\omega} t[\hat{m}]$) will be assigned a gain value greater than zero. Hence, the heuristic will not select any $r \in R$ again as long as such r_ω exists.

For instance, after r_1 is added in the flight record example, the rule (London, *, *) has the highest information gain out of all possible rules according to $gain(r)$ above.

2.5 Multidimensional Data Mining using Cube Lattices

Recall from Section 2.1 that a rule is essentially an element in the multidimensional space $(dom(A_1) \cup \{*\}) \times \cdots \times (dom(A_d) \cup \{*\})$. Following the notations in multidimensional data mining, we further define the generalization/specialization order between rules to establish the structure of a *cube lattice* [9]. A rule r is an *ancestor* of another rule r' if and only if it satisfies the following property: $\forall A_i \in \{A_1, A_2, \cdots, A_d\}$, either $r[A_i] =$ " $*$ " or $r[A_i] = r'[A_i]$. For convenience, we also denote r' as a *descendant* of r. By definition, every rule in the cube lattice is its own ancestor and descendant. The immediate proper ancestors of a rule are the *parent* rules while the immediate proper descendants are the *child* rules. For a given tuple t, we use $CL(t)$ to represent the cube lattice of a multidimensional database relation γ containing a single tuple t.

For example, consider a multidimensional database relation γ containing only the first tuple in Table 1.1. Figure 2.1 shows the cube lattice of γ. The elements of the cube lattice correspond to the cuboids 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

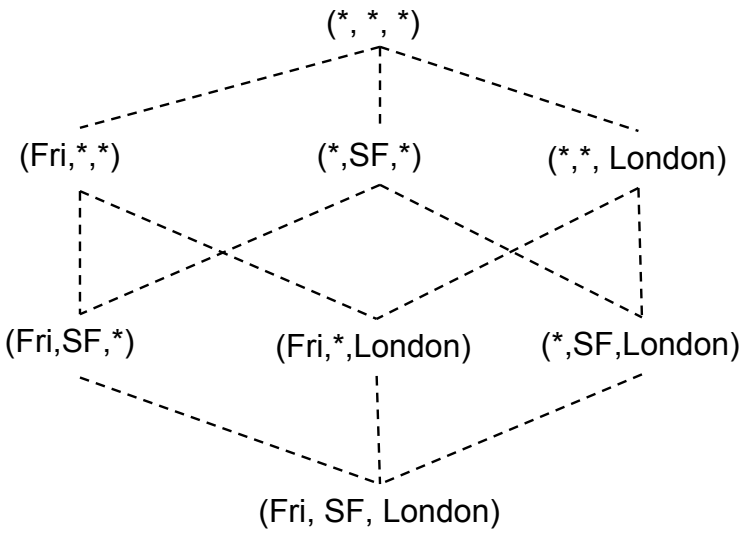

Figure 2.1: Cube lattice for a multidimensional database relation with a single tuple (Fri, SF, London)

wildcard, and so on up to the $(d + 1)$-th level, where d is the number of dimension attributes. Elements in the higher levels are the ancestors of those from the lower levels that are connected to them through a path. For example, (*, *, *) is the only ancestor of (*, SF, *).

2.6 Data Processing Platforms

In this section, we introduce several alternatives for the data processing platform to support SIRUM.

2.6.1 PostgreSQL

PostgreSQL is an open source object-relational database system, notable for its extensibility, advanced features and standards-compliance. PostgreSQL's support for user-defined functions and customized data types is critical for the implementation of SIRUM. Notice that computing the cube lattice involves generating all possible ancestors from a base tuple or rule. It is challenging to efficiently compute the result if only standard SQL queries are available.

PostgreSQL is capable of storing and processing large-scale relational data sets [26]. For the analytical queries used in large scale informative-rule rule mining, however, the major scalability limitation for PostgreSQL is its lack of built-in support for intra-query parallelism. The evaluation of algorithms in [16] only uses a single database session, which is limited to a single process in PostgreSQL to execute the SQL queries. Moreover, the process cannot utilize more than one CPU.

Another hurdle to scaling informative rule mining on PostgreSQL is its emphasis on optimizing disk-based access. But there is a potential performance gain by keeping frequently accessed data in memory. For instance, caching the estimate values in memory during iterative scaling reduces the amount of random write access to the disk.

Due to the limited scalability of PostgreSQL in dealing with analytical workloads, we consider other alternative platforms, as discussed below.

2.6.2 Apache Hive

Apache Hive [33] is an open source Hadoop-based data warehouse infrastructure for processing and managing large data sets on distributed storage systems such as Apache HDFS. It provides an SQL-like query language, HiveQL, to facilitate data analysis. The HiveQL compiler translates query statements into a directed acyclic graph of jobs and submits them to Hadoop YARN [34], a resource management infrastructure, for execution. Currently, Hive supports three execution engines: Apache Tez [28], Apache Spark [38], and MapReduce [14]. In this thesis, we only use MapReduce to evaluate HiveQL queries.

2.6.3 Apache Spark

Apache Spark [38] is another open source cluster computing framework designed for large scale data processing. At the core of Spark is a distributed memory abstraction called resilient distributed data set (RDD) [37]. The corresponding application programming interface is restricted to coarse-grained transformations such as map and join for efficient implementation of fault tolerance. Compared to MapReduce, Spark provides a more efficient data sharing abstraction in terms of data replication and serialization costs and support a wider range of cluster programming models.

In section 5.2, we will experimentally compare the three platforms above. For future work, we are planning to explore other options such as Tez and Cloudera Impala.

Chapter 3

Profiling Baseline Implementations

In this chapter, we will start with a description of the baseline implementation of informative rule mining, followed by a profiling study over the baseline implementation to uncover the performance bottlenecks and opportunities for improvements. We will address the bottlenecks and exploit the opportunities in Chapter 4.

3.1 Naive SIRUM

In this section we describe a naive implementation of SIRUM. At the high level, Naive SIRUM is a greedy algorithm iteratively computing the best rule and adding it into the rule set R, as shown in Algorithm 2. To begin with, Naive SIRUM first adds the rule that covers all tuples, $r_1 = (*, *, ..., *)$, into R. Then it performs iterative scaling to adjust $\lambda(r)$ such that $\sum t[m] = \sum t[\hat{m}]$ for all $t \asymp r_1$. Moving on to the next iteration, Naive SIRUM computes the information gain of all possible rules, selects the one with the highest gain, r_2, and add it to R, followed by another round of iterative scaling to adjust $\lambda(r)$ such that $\sum_{t \asymp r} t[m] = \sum_{t \asymp r} t[\hat{m}]$ for every $r \in R$. For the following iterations, Naive SIRUM repeats the procedure above until r_{k+1} is added to R. Note that a total of k rules are generated because the r_1 is added as the first rule by default. In each iteration, we distinguish the two major steps as *candidate rule generation* and *iterative scaling*.

The rule generation step follows a simple MapReduce-based data cube algorithm presented in [25][1]. The group-by attributes are subsets of $\{A_1, \cdots, A_d\}$ and the aggregate functions are

[1]The paper also presents an improved algorithm for holistic measures but the simple algorithm is shown to work as well as the improved algorithm for algebraic measures such as the one in this thesis

$SUM(t[m])$ and $SUM(t[\hat{m}])$. Each element in the cube lattice corresponds to a candidate rule whose gain is $SUM(t[m]) \times \log \frac{SUM(t[m])}{SUM(t[\hat{m}])}$.

The data cube algorithm starts by launching mappers to process tuples of the input dataset in parallel. For each tuple $t \in D$, the mapper emits a key-value pair for each element in the cube lattice of t, where the key is the element and the value is a tuple $(t[m], t[\hat{m}])$. The mapper output is then shuffled and delivered to the reducers. Each reducer processes a subset of the key-value pairs: for each distinct key (i.e., each element in the cube lattice), computes the gain from the partially-aggregated values produced by the mappers or combiners. The element with the highest gain is chosen to be the most informative rule and added to R.

Algorithm 2 Computing Rule Set with Naive SIRUM

Input: D, k
Output: R

1: Add (*, $\cdots$, *) to R
2: **for** $i \leftarrow 1$ to k **do**
3: Identify the rule with the highest information gain, r_i, from a set of candidate rules C_i and add it to R
4: Update the current knowledge of distribution of m based on R using iterative scaling
5: **end for**

3.1.1 Sample-based Candidate Pruning

The set of all possible rules grows exponentially with respect to d, the number of dimensions. Observe that the gain formula (Equation 2.2) does not satisfy the downward closure property [4], a commonly used technique in association rule mining and to efficiently prune the search space. Specifically, an ancestor rule being not informative does not imply that any of its descendants is not informative. It follows that the ancestor rule and all of its descendants cannot be pruned as the algorithm traverses through the cube lattice.

One way of addressing the deficiency above is to draw a random sample s from D and only consider rules in the cube lattice of s [16]. The intuition is that rules with frequently-occurring combinations of dimension attribute values are likely to appear in s and are more likely to have high gain. we refer to this approach as *sample-based candidate pruning*, as opposed to *exhaustive candidate exploration* that computes the gain of every possible rule.

To distinguish the rules selected by the two approaches, we refer to the one chosen by exhaustive candidate exploration as the most informative rule (MIR). The value of $|s|$ dictates the

size of the partial cube lattice from which the algorithm searches for a rule with the highest gain. Intuitively, the best rule selected from a larger group of candidates is more likely to have a gain closer to the gain of the MIR. We consider the value of $|s|$ to be *sufficiently large* if the KL-divergence of the eventual rule set is close to the one produced using exhaustive candidate exploration.

The algorithm starts by drawing a random sample s from D and computing a cross-product of s and D. For each pair of joined tuples, it outputs a rule that is the least common ancestor (LCA) of the tuples, as defined in Section 2.1. We denote the result as $LCA(s, D)$, which is stored in an RDD. Next, it emits all possible ancestors of the LCAs (including the LCA itself) and their aggregate values in the map stage. In the reduce stage, it computes the information gain for the candidate rule set, which only contains the LCAs and their ancestors. Then, it computes the cross-product of the candidate rule set and s to adjust the aggregate values. If a candidate rule, r, matches more than one tuple in s, the corresponding tuple in D must have contributed its aggregate value to r more than once. It follows that the aggregate value of r should be divided by the number of matching tuples in s. Finally, the rule with the highest gain is added to R. In the remainder of this thesis, unless otherwise specified, Naive SIRUM includes sample-based candidate pruning.

For instance, suppose we sample two tuples from Table 1.1 and we get t_4 and t_9. Projected onto their dimension attributes, these are (Sun, Chicago, London) and (Thu, SF, Frankfurt). For each tuple in the sample, we compute the LCA of it and each tuple in D, where we replace non-matching attribute values by stars. For example, the LCA of (Sun, Chicago, London) and (Fri, SF, London) is (*, *, London). Using t_4 and t_9 as the sample, we get the following LCAs: (*, *, *), (*, *, London), (*, *, Frankfurt), (*, Chicago, *), (*, SF, *), (Sun, *, *), (*, SF, Frankfurt), (Sun, Chicago, London) and (Thu, SF, Frankfurt). Next, we generate all the ancestors of each LCA, and use the LCAs and their ancestors as the 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.

Now we analyze the complexity of sample-based candidate pruning. Computing $LCA(s, D)$ requires joining s and D, which means the complexity of this step is $O(|s| \cdot |D|)$. Let $P(s)$ be the set of all possible ancestors of tuples in s. The second and third step takes $O(|P(s)| + |P(s)| \cdot |s|)$ operations to compute. Choosing the top rule in the final step requires only a single scan and the cost is $O(P(s))$. Therefore, the overall complexity is $O(|s| \cdot |D| + |P(s)| \cdot |s|)$.

3.2 BJ SIRUM

In this section, we describe an improvement to the implementation of Naive SIRUM on distributed data processing platforms (i.e., Spark, Hive).

The key observation is that the number of tuples in R and s are much smaller than that in D. It allows us to use Spark's broadcast join [8, 10], i.e., map-side join, to compute the LCAs (which requires s joining D) and to compute lines 3-6 of the iterative scaling algorithm (which requires R joining D). Specifically, we create multiple copies of the smaller datasets on 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. The performance gain stems from the observation that shuffling the partitions of the larger dataset is likely more I/O-intensive than replicating the smaller dataset to all workers. We refer to this implementation as BJ SIRUM (Broadcast Join SIRUM), which is the implementation we use for the profiling results below. Unless otherwise specified, BJ SIRUM includes sample-based pruning because it improves upon Naive SIRUM.

In the remainder of this thesis, unless stated otherwise, BJ SIRUM is the baseline implementation in experimental evaluations.

3.3 Profiling BJ SIRUM

In this section, we present preliminary experiment results of the baseline implementation (BJ SIRUM) to reveal performance bottlenecks and provide guidance for improvement. In Chapter 4, we will present performance optimizations based on the insights below.

Similar to Naive SIRUM, BJ SIRUM also has two major steps: candidate rule generation (including gain computation) and iterative scaling. We measure the execution time of both steps using the following real datasets that we will describe in Chapter 5: `Income`, `GDELT`, and `SUSY`. The `Income` dataset has approximately 1.5 million tuples, 9 dimension attributes and a binary measure attribute; The `GDELT` dataset has around 3.8 million tuples, 9 dimension attributes and a numeric measure attribute; the `SUSY` dataset has 5 million tuples, 18 dimension attributes and a binary measure attribute; and TLC has 160 million tuples and 9 dimension attributes. The experimental setup of our profiling results is described in Section 5.1.

Figure 3.1 shows the execution time of BJ SIRUM for each dataset as well as the total execution time. A sample of 64 tuples are drawn from D to perform sample-based candidate pruning and 10 rules, in addition to the rule with all wildcards, are added to R by the end of each experiment. We set $|s|$ to 64 because it is sufficiently large (as defined in Section 3.1.1) for all three

data sets. It is also a recommended sample set size for the Income dataset in [16]. As shown in the plot, both rule generation and iterative scaling account for a noticeable amount of overall execution time but their weights vary depending on the dataset. One of the key observations is that the major performance bottleneck shifts from iterative scaling to rule generation as the number of dimension attributes increases from 9 to 18. We also experimented with a larger value of $|s|$, which causes the execution time of rule generation to increase. For a sufficiently large value of $|s|$, however, iterative scaling remains the major bottleneck for Income and GDELT. The total runtime is highest, by far, 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.

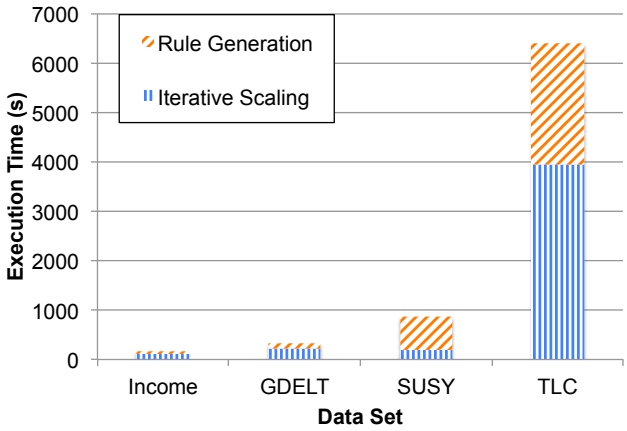

Figure 3.1: Baseline SIRUM runtimes (k = 10, $|s|$ = 64)

Now we dive into the workflow of candidate rule generation as the number of dimension attributes increases. Specifically, candidate rule generation involves the following three steps:

1. Candidate pruning, i.e., computing the cross-product of s and D to get the LCAs.
2. Generating the ancestors of the LCAs by the mappers.
3. Computing the information gain of each LCA and its ancestors by the reducers.

We use the Income, GDELT datasets as well as projections of SUSY on the first 10, 14 and 18 dimension attributes. Figure 3.2 shows the relative and absolute execution time of BJ SIRUM over the five datasets. First, observe that computing the information gain is relatively inexpensive after the candidate rules are generated. But the cost increases if the number of LCAs and

their ancestors grows. On the other hand, candidate pruning is the bottleneck for datasets with relatively few dimension attributes, accounting for over 90 percent of the rule generation runtime for `Income` and `GDELT`. For datasets with more dimension attributes, however, the ancestor generation step becomes the bottleneck even though we are using sample-based candidate pruning. The behaviour can be explained by the exponential growth in the number of ancestors with respect to the number of dimension attributes. The runtime of gain computation step follows a similar trend because it computes information gain for all rule output by the ancestor generation step.

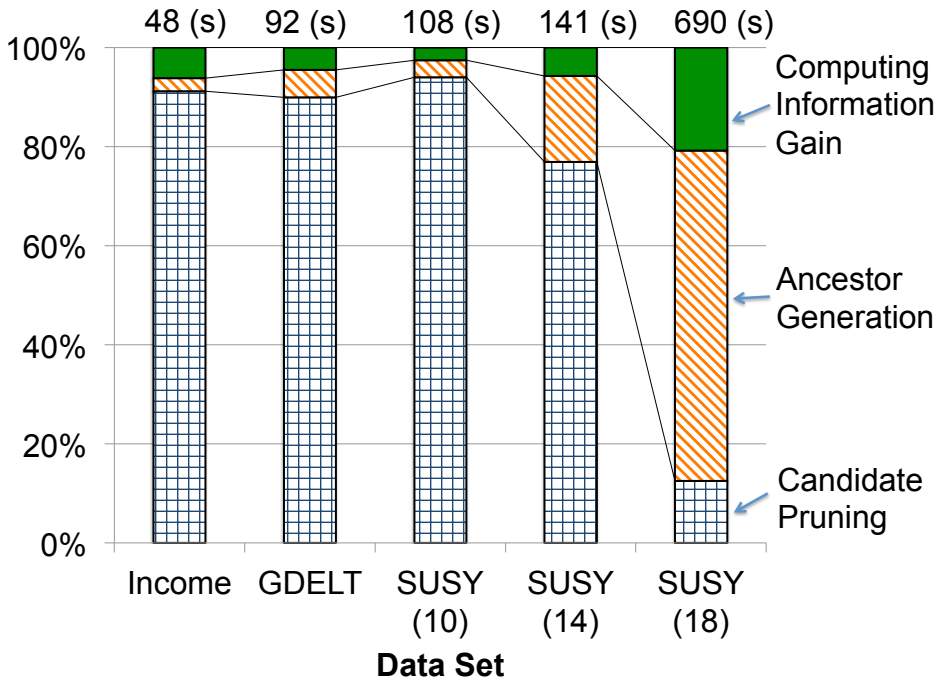

Figure 3.2: Rule generation runtimes by step (k = 10, $|s|$ = 64)

Motivated by these observations, in the next chapter we present performance improvements that target each of iterative scaling, candidate pruning and ancestor generation.

Chapter 4

Performance Improvements

In this chapter, we describe the following performance improvements over BJ SIRUM:

- A strategy to maintain only the necessary information for iterative scaling and avoid repetitive access to the input data set (Section 4.1)
- An inverted index to accelerate the computation of LCA rules (Section 4.2)
- A multi-stage ancestor generation pipeline that significantly reduces the overall output size from the mappers (Section 4.3)
- Adding multiple disjoint rules at a time to reduce the total execution time (Section 4.4)
- Reducing disk I/O for datasets failing to fit in the main memory of the computing cluster

4.1 Fast Iterative Scaling

Recall that in Algorithm 1 we present the workflow of iterative scaling, which is executed whenever a new rule is added into R. In the profiling experiments of BJ SIRUM, iterative scaling had to loop for approximately 10 times on average before the multipliers converged. Moreover, scaling the multiplier of one rule can trigger other rules to scale their multipliers in order to restore the balance between $\sum_{t \succeq r} t[m]$ and $\sum_{t \succeq r} t[\hat{m}]$, if their support sets overlap. It means the number of iterations tends to increase as the size of R grows. Hence, improving the performance and efficiency of iterative scaling is imperative to reducing the overall execution time of SIRUM.

From Algorithm 1, observe that D is accessed twice per iteration: the first access happens when computing $\hat{m}(r_i)$ at lines 3-6 and the second access happens while updating $t[\hat{m}]$ at line 11. Furthermore, when accessing D, the $t \asymp r$ condition is evaluated 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, Algorithm 3 caches the results of $t \asymp r$ for previously added rules. It maintains a bit array BA for each tuple t, whose i-th entry, $t.BA[i]$, is one if $t \asymp r_i$ is true and zero otherwise. For instance, in Table 1.1, t_1 has $t.BA = 1100$ because it matches r_1 and r_2 from Table 1.2. Algorithm 3 also associates a bit array with each rule $r_i \in R$, $r_i.BA$, which has a one in the ith entry and zeros elsewhere. When a new rule r_w is added to R, Algorithm 3 updates 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 $r_i.BA$ rather than attribute-by-attribute comparison.

Next, Algorithm 3 avoids repeatedly accessing D during iterative scaling. Recall that for a given tuple t, $t[\hat{m}]$ is a product of the multipliers λ of each rule that matches it. Observe that tuples with the same BA, i.e., those which match the same rules, have exactly the same $t[\hat{m}]$, namely $\prod_{i,t.BA[i]=1} \lambda(r_i)$. Now, return to Table 1.1 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, to which we refer as a Rule Coverage Table (RCT), is shown in Table 4.1. 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 1.1. Also, note that the first bit of every BA is one since every tuple matches the all-wildcards rule.

Algorithm 3 Iterative Scaling with RCT

Input: D, R, λ, $m(r)$ for all $r \in R$, ϵ, newly added rule r_w

for $t \in D$ **do**
  if $t \asymp r_w$ **then**
    $t.BA[w] \leftarrow 1$ # $t.BA[w]$ default to 0
  end if
end for
Group by $t.BA$ and aggregate over $COUNT(*)$, $SUM(t[m])$, and $SUM(t[\hat{m}])$ to compute the RCT
while true **do**
  $DIFF \leftarrow ARRAY(|R|, 0)$
  for $r_i \in R$ **do**
    $\hat{m}(r_i) \leftarrow \frac{\sum_{p \in RCT, r_i.BA \& p.BA \neq 0} p.SUM(t[\hat{m}])}{\sum_{p \in RCT, r_i.BA \& p.BA \neq 0} p.count}$
    $DIFF[i] \leftarrow \frac{|m(r_i) - \hat{m}(r_i)|}{|m(r_i)|}$
  end for
  $next \leftarrow \underset{i}{\text{argmax}} DIFF[i]$
  if $DIFF[next] > \epsilon$ **then**
    $\lambda_{old} \leftarrow \lambda(r_{next})$
    $\lambda(r_{next}) \leftarrow \lambda(r_{next}) * \frac{m(r_{next})}{\hat{m}(r_{next})}$
    for $p \in RCT$ **do**
      if $p.BA \ \& \ r.BA \neq 0$ **then**
        $p.SUM(t[\hat{m}]) \leftarrow p.SUM(t[\hat{m}]) * \frac{\lambda(r_{next})}{\lambda_{old}}$
      end if
    end for
  else
    for $t \in D$ **do**
      $t[\hat{m}] \leftarrow \prod_{r \in R, t \asymp r} \lambda(r)$
    end for
    break
  end if
end while

Each row of the RCT in Table 4.1 describes a subset of tuples from D that is pairwise disjoint

with every other row of the RCT, as illustrated in Figure 4.1. 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.

Table 4.1: 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

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 $SUM(t[\hat{m}])$ by the sum of $count$ of these rows (line 10 in Algorithm 3). Then, after scaling $\lambda(r_{next})$ in line 16, instead of accessing D to modify the affected $t[\hat{m}]$s, 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 to 21). 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 23-25).

For the rule set to be interpretable by the data analyst, the maximum number of rules in R is expected to be no more than 50. Moreover, the amount of reduction in KL-divergence per rule tends to decrease as $|R|$ grows because rules with higher gain are typically added to R first. It means RCT is likely to be much smaller than D and can be replicated on each worker node, the runtime of iterative scaling can thus decrease significantly (as we will show in Chapter 5).

Now we analyze the space efficiency of RCT. If $|R| \leq 50$, RCT only requires a bit map of at most 50 units for each $t \in D$ and $r \in R$. It is a small overhead compared to the storage requirement of the dimension and measure attribute values.

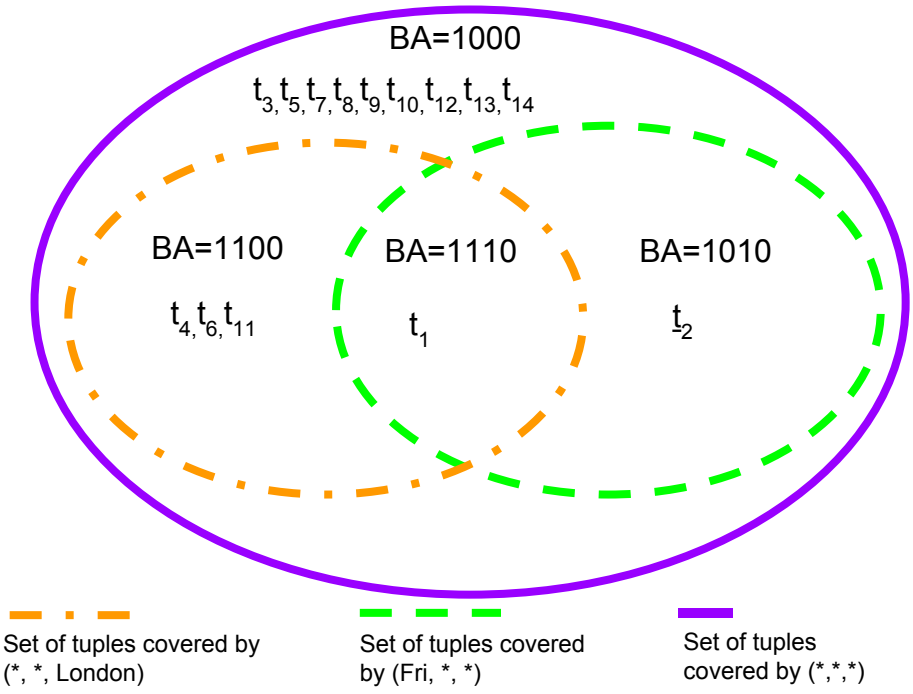

Figure 4.1: Illustration of the RCT from Table 4.1

4.2 Fast Candidate Pruning

We now move to optimizing candidate rule processing. Recall from Section 3.3 that candidate rule pruning and ancestor generation are the two expensive operations. In this section, we start with improving the performance of candidate rule pruning.

Sample-based candidate pruning (Section 3.1.1) computes the LCA of each tuple from the sample s with each tuple from the dataset D. For each LCA, it makes d comparisons, one for each dimension attribute, to decide whether that attribute value of the LCA should be a wildcard, which happens when the two tuples have different values of this attribute or a constant, which happens when the values match. If s is much smaller than D and the distribution of attribute value is not dominated by a single constant, then a particular attribute value of an LCA is more likely than not to be a wildcard. This suggests a simple optimization: rather than computing the LCA via a cross product of s and D, initialize all $|s||D|$ LCAs to all-wildcards and then replace wildcards with constants where necessary. If we can quickly locate the constants in the LCAs, i.e., when the two tuples involved in the LCA agree on at least one attribute, we should be making

fewer than d comparisons per LCA on average, thereby improving performance.

One solution to locate the positions of the constants is to pre-process s and build an inverted index for each attribute, where each entry in an index consists of an attribute value and a list of pointers to the tuples in which that value occurs in this particular attribute. With this index replicated as a broadcast variable, each mapper can exploit it as follows. For each tuple t in the fragment of D stored at this mapper, the mapper initializes a buffer of $|s|$ LCAs with each default to all-wildcards (each tuple from D produces exactly $|s|$ LCAs, one for each tuple in s). Note that the i-th LCA in the buffer corresponds to the output of the i-th tuple in s. Then, for each dimension attribute A_i, it uses the index to look up the pointers to tuples $t_s \in s$ with $t_s[A_i] = t[A_i]$. Finally, it identifies the corresponding LCAs in the buffer and updates their A_i attributes to $t[A_i]$.

For example, suppose we sample two tuples from Table 1.1 and we get t_4 and t_9. An index over the sample has an entry whose key is 'Sun' and the value contains pointers to t_4. When $t_3 \in D$ is joined with s, a buffer is first created with the following form: [(*, *, *), (*, *, *)]. Note that the t_3's value in the 'Day' attribute matches the index key 'Sun' and the index value points to the first tuple in the sample (t_4). We can update the buffer to [(Sun, *, *), (*, *, *)]. Hence, it requires 3 look-up operations rather 6 attribute-by-attribute comparison to output the list of LCAs.

We now provide a brief analysis of the performance gain of fast candidate pruning. Let t_i be a tuple from D and s_j be a tuple from the sample s. We will save one comparison operation whenever $t_i[A_k] \neq s_j[A_k]$. Let $dom(A_k) = \{V_1, V_2, ..., V_m\}$ and the relative frequency of V_n in D be F_n. Assuming that s has approximately the same data distribution as D and that the occurrence of $t_i[A_k]$ and $s_j[A_k]$ is independent, the probability that $t_i[A_k] = s_j[A_k] = V_n$ is ${F_n}^2$. Thus, the probability that $t_i[A_k] \neq s_j[A_k]$ is $1 - \sum_{n \in [1,m]} {F_n}^2$. Let $h = \underset{n}{\operatorname{argmax}} F_n$. Then $1 - \sum_{n \in [1,m]} {F_n}^2 \geq 1 - {F_h}^2 - (1 - F_h)^2$. Thus, even for a skewed dataset where $F_h = 0.9$, we can still obtain an 18 percent improvement, and the improvement becomes more significant if the distribution of values of A_k is less skewed.

4.3 Fast Candidate Rule Processing

In Section 3.3, the profiling results reports that as the number of dimension attributes increases, ancestor generation becomes the bottleneck, even with sample-based candidate pruning. This is because the simple data cube algorithm used by BJ SIRUM generates all the ancestors of each LCA and computes the gain of each candidate rule, all in one map-reduce round. Even though the number of distinct ancestors in the final output from the reducer is relatively small, the

mapper emits as intermediate result a much larger number of key-value pairs before aggregation or combination takes place. A large size of *intermediate* result leads to high CPU usage of the mappers as well as high memory usage, resulting in stragglers, additional CPU overhead due to garbage collection to free up memory, or even disk I/O.

To address this problem, we can split up the processing of candidate rules into multiple map-reduce rounds, with each round generating a subset of the ancestors along the cube lattice. The ancestors generated in the current map-reduce round will become the input to the next round to generate senior ancestors, which is a more efficient way to produce candidate rules, as we will elaborate below. This way, it reduces the load on both the mappers and reducers in each round, but it incurs the overhead of starting up new map-reduce operation. Thus, it is also important to limit the number of map-reduce rounds.

To split up the computation, we propose a simple extension of the distributed data cube algorithm. First, we compute the LCAs, as before. Next, we randomly partition the dimension attributes into g ordered parts: $GrpList = (G_1, G_2, G_3, \ldots, G_g)$. Each part corresponds to a separate map-reduce stage. For the first group, we take the LCAs and have the mappers generate the subset of their ancestors that have wildcards in the attributes from G_1. Next, the reducers compute the gain only for this subset of candidate rules. In the second stage, we take in the output of the first stage, namely the subset of elements from the cube lattice which has already been computed, and we generate (and compute the gain of) all their ancestors that have wildcards in the attributes from G_2, and so on. At the end, we will have computed all the tuples in the data cube corresponding to the LCAs and all their ancestors, i.e., all the candidate rules.

We illustrate the procedure using an example. Recall Table 1.1 and suppose that (Fri,SF,London) is an LCA. BJ SIRUM performs a single map-reduce round. For each LCA, the mapper emits all of its ancestors. Thus, for (Fri,SF,London), it generates all the ancestors shown in Figure 2.1. The reducers then compute the gain for each candidate rule, i.e., each LCA and all of its ancestors. Now suppose we partition the dimension attributes into two groups: G_1=$\{Day, Origin\}$ and G_2=$\{Destination\}$. In the first map-reduce round that deals with G_1, when a mapper processes (Fri,SF,London), it only generates its ancestors that have wildcards in the Day or Origin attributes, but not Destination. That is, the generated ancestors are: (Fri,*,London), (*,SF,London), (*,*,London), as shown in Figure 4.2 and labelled "G1". The reducers then compute the gain of the LCAs and their ancestors generated in this round. Next, in the second round, we receive all the rules computed in the first round as input, and compute the gain of the remaining candidate rules having a wildcard in the G_2 attributes (labelled "G2" in Figure 4.2). When a mapper encounters (Fri,SF,London) in this round, its gain will already have been computed. The mapper generates only one ancestor, (Fri,SF,*), whose aggregate values correspond to $\sum t[m]$ and $\sum t[\hat{m}]$ of (Fri,SF,London). Similarly, the mapper responsible for (Fri,*,London) only generates the ancestor (Fri,*,*), the mapper responsible for (*,SF,London) only generates the ancestor

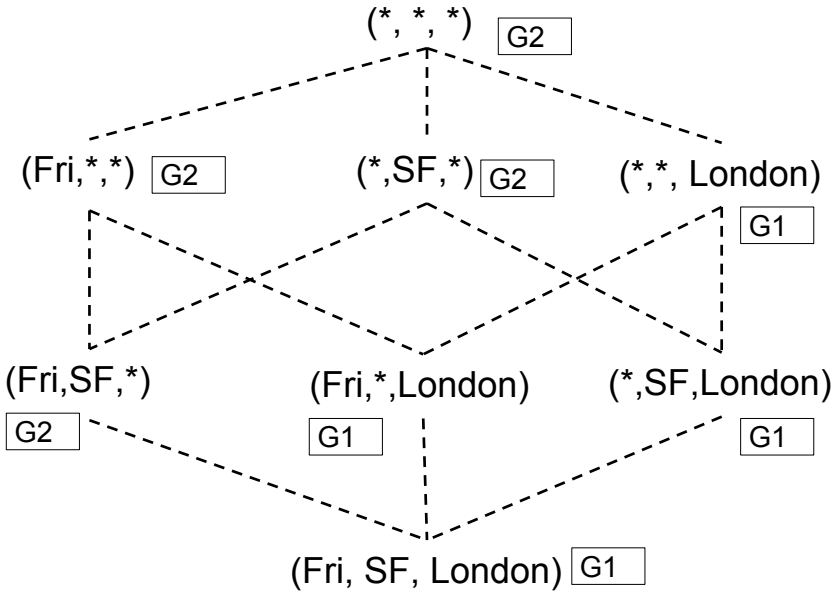

Figure 4.2: Computing candidate rules in two stages

(*,SF,*), and the mapper responsible for (*,*,London) generates (*,*,*). The reducers then compute the gain for these candidate rules.

The extension reduces the total number of key-value pairs emitted by the mappers. Consider two LCA rules r_i and r_j. Suppose that they share a common ancestor r_ω. Let r_θ be any senior ancestor of r_ω that is generated in a later stage than r_ω. For r_i and r_j, the total number of ancestors emitted by the mappers in BJ SIRUM is $CL(r_i) + CL(r_j)$ in which both r_ω and r_θ are emitted twice, once from $CL(r_i)$ and once from $CL(r_j)$. With the extension, r_ω is still emitted twice but r_θ is only emitted once because it is generated from $CL(r_\omega)$.

Now we use an example to illustrate the benefit this column grouping idea. Consider the following two LCAs, (Mon, SF, London) and (Fri, SF, London). They share the ancestor (*, SF, London) in common. Instead of emitting (*, SF, *) twice from the cube lattices of (Fri, SF, London) and (Mon, SF, London), the extension allows (*, SF, *) to be emitted only once from the cube lattice of (*, SF, London).

We provide a detailed study on the correctness of the column grouping in Appendix A.

4.4 Generating Multiple Rule per Iteration

The optimizations in previous sections have focused on speeding up particular steps of SIRUM. In this section, we present a way to reduce the total number of steps: rather than choosing the most informative rule per iteration, choose multiple rules from the top of the list sorted by information gain. To simplify the analysis, we consider the case where two rules are chosen per iteration. This could potentially cut down the runtime by a half, independently of the other optimizations, since half as many rule generation and iterative scaling runs are needed.

Notice that the second most informative rule in the i-th iteration may not be among the most informative rules in the $(i + 1)$-st iteration, after the best rule from the i-th iteration has been added. Specifically, suppose the support set of the second most informative rule overlaps with that of the first one. Its information gain may drop drastically after adding the first rule into the rule set. We can help avoid this problem by choosing the next most informative rule that is *disjoint* from the most informative one; if the top rule is added, any rules that overlap with it will have their information gain changed, but updating the information gain amounts to doing iterative scaling and recomputing the $\hat{m}(r)$s, which brings us back to choosing one rule per iteration. Furthermore, SIRUM can demand that the next most informative rule has a sufficiently high information gain—say, at least half the gain of the top rule—and/or is among the top, say, one percent of the rules in the current iteration. Even then, if SIRUM performs $\frac{k}{2}$ iterations of the rule mining process and choose two rules at a time, it may end up with a higher KL Divergence compared to choosing one rule at a time k times. Following a similar strategy, one can extend it to choose three or more rules per iteration, as long as the rules are mutually disjoint. We will experimentally explore this optimization in Chapter 5.

For example, suppose the top rule is (*, SF, *), followed by (Fri, SF, *) and (*, London, *). By definition, the second best rule, (Fri, SF, *), overlaps with (*, SF, *) while the third best rule is disjoint from (*, SF, *). After selecting (*, SF, *) into R, SIRUM can also add (*, London, *) into R before proceeding to the next iteration.

4.5 Scaling towards Very Large Datasets

Even with the optimizations from the previous sections, SIRUM still requires access to D twice per iteration: once to compute the LCAs in the candidate pruning step and once when updating the bit arrays and computing the RCT. If D fails to fit in the main memory of the cluster, the performance penalty of HDFS I/O may be significant. In Spark, this problem is further exacerbated 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 memory allocated to the executor: 3GB and 5GB. The rest of the experimental setup is specified in Section 5.1. Only one worker node is used in this experiment; the findings are similar with more nodes, as we will present in Section 5.7.3. 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 4.3 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 inadequate 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.

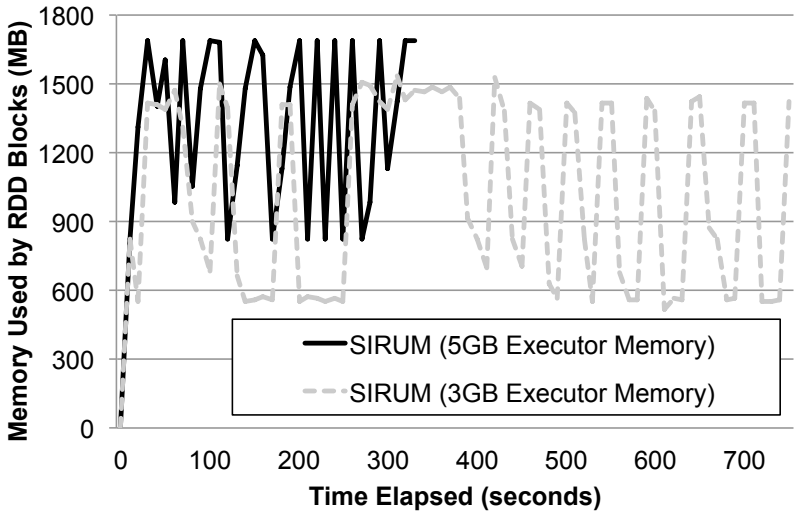

Figure 4.3: Memory usage over time: different memory allocations

Motivated by the need to reduce disk I/O, we design a strategy referred to as *SIRUM on sample data* for very large datasets, in which we draw a random sample of D of size determined by the amount of memory available, and use it instead of D during rule generation and iterative scaling. In Figure 4.4, 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 sampled datasets fit in memory (no disk I/O after they are fully loaded) 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

examine this trade-off in Chapter 5.

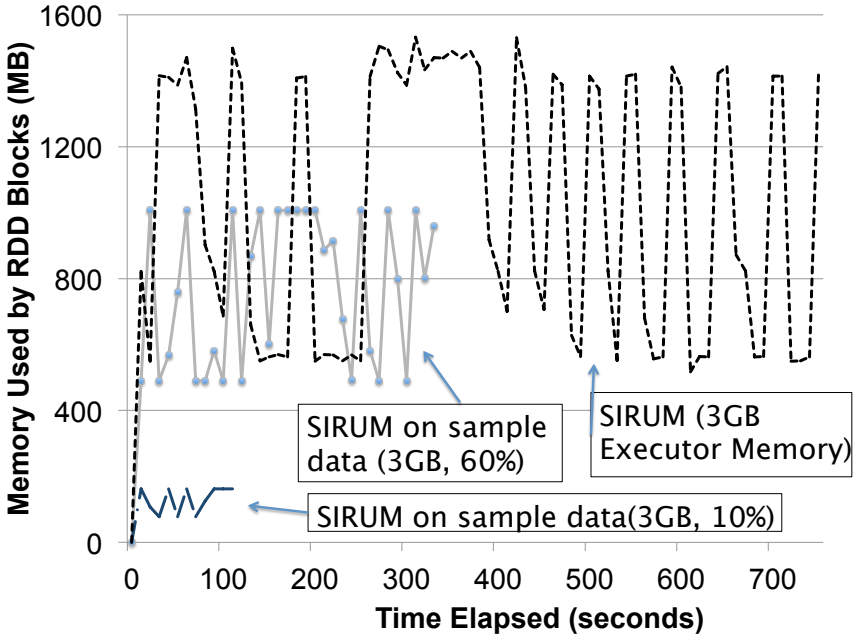

Figure 4.4: Memory usage over time: SIRUM vs. SIRUM on sample data

4.6 Summary

To evaluate the effect of optimizations covered in this chapter, we implement multiple SIRUM variants based on BJ SIRUM, as listed in Table 4.2. Each variant incorporates a single optimization, except for Optimized SIRUM that combines the optimizations in order to achieve the best performance. Note that SIRUM on sample data is a special measure to deal with datasets larger than the memory capacity. Since it is not applicable when the dataset does fit in memory, it is not considered a variant of SIRUM.

Table 4.2: SIRUM Variants

Name	Base Implementation	Optimizations
Baseline SIRUM (BJ SIRUM)	Naive SIRUM	Broadcast Join
RCT SIRUM	Baseline SIRUM	Rule Coverage Table
FastPruning SIRUM	Baseline SIRUM	Fast Candidate Pruning
FastAncestor SIRUM	Baseline SIRUM	Fast Candidate Rule Processing
Multi-rule SIRUM	Baseline SIRUM	Multi-rule Insertion
Optimized SIRUM	Baseline SIRUM	Rule Coverage Table Fast Candidate Pruning Fast Candidate Rule Processing Multi-rule Insertion

Chapter 5

Evaluation

In this chapter we present our experimental results. We start by describing the experimental environment and datasets used in Section 5.1. 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 5.2). From Section 5.3 to 5.5, we show the performance improvement of the optimizations from Chapter 4. Then, we show the performance improvement of optimized SIRUM for informative rule mining and data cube exploration (Section 5.6) compared to straightforward distributed implementations of previous work [16, 29]. Finally, we evaluate the scalability of optimized SIRUM and SIRUM on sample data in Section 5.7.

5.1 Experimental Setup

5.1.1 Experiment Environment

The evaluation of SIRUM was performed on a cluster of 16 nodes, each running CentOS 6.4 with 4 Intel Xeon E5-2620 2.10 GHz 6-core CPUs, 64 GB of DDR3 RAM and 2.7 TB of local disk space. The cluster also has the following software packages installed: (1) Apache Hadoop 2.6; (2) Apache Spark 1.4.0; (3) PostgreSQL 9.4.4; and (4) Apache Hive 1.2.0.

We implemented the SIRUM variants listed in Table 4.2 on Apache Spark, which is chosen as the ideal underlying platform based on the performance results presented in Section 5.2. They were deployed as Spark applications in cluster mode on Hadoop YARN, the resource management module in the Hadoop framework. If not stated otherwise, the configurations below are used throughout this chapter:

1. 16 Spark executors are launched with one per node.
2. 45 GB of memory is allocated to each executor. The other 19 GB on each node is reserved for the operating system, Spark overhead, and the driver program (if needed), etc.
3. 8 GB of memory is allocated to the driver program, which resides on one of the cluster nodes along with the executor, running inside the Application Master process managed by YARN.
4. The input data RDD is initially divided into 384 partitions with each partitioned assigned to a Spark task.
5. The threshold for iterative scaling, ϵ, is set to 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-wildcards rule minus the KL-divergence using the given set of rules.

5.1.2 Data sets

We use the following data sets to evaluate the effect of optimizations. All datasets are stored as CSV files in HDFS with a replication factor of 3.

- `Income` contains U.S. Census data with household demographic attributes such as the number of children and marital status, and a binary measure attribute to indicate if the given household's income exceeds $\$100,000$. This data set 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 that does not [6]. 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 [23] which records over 300 categories of events around the world such as riots and diplomatic exchanges, with nearly 60 attributes including location and actor backgrounds. 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 was accessible through `www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`. The dataset occupies 8.5 GB in HDFS.

We used the `TLC` dataset to evaluate the scalability of SIRUM with respect to the number of data tuples. First, a sample of 160 million tuples, denoted by *TLC_160m*, were randomly drawn from the full data set. *TLC_160m* is the largest sample dataset that can be entirely stored in the allocated main memory of the cluster. Similarly, we drew another random sample of 80 million tuples, denoted by *TLC_80m*, from *TLC_160m* and so on. The `TLC` dataset itself is only used in Section 5.1 where we examine the performance of SIRUM as it takes as input a sample from a very large data set to avoid frequent disk access, trading a small drop in accuracy for better performance.

5.2 Choice of Data Processing Platforms

In Section 2.6, we discussed three candidates of data processing platforms to support the implementation of SIRUM: PostgreSQL, Apache Hive, and Apache Spark. In this section, we compare Baseline (includes sample-based pruning and broadcast join) implemented on these platforms using real data sets to justify our choice of Spark as the underlying platform.

Figure 5.1 compares the runtime of Baseline SIRUM on Spark and PostgreSQL using a single compute node with `Income`, whereas Figure 5.2 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 lies in the disk and network

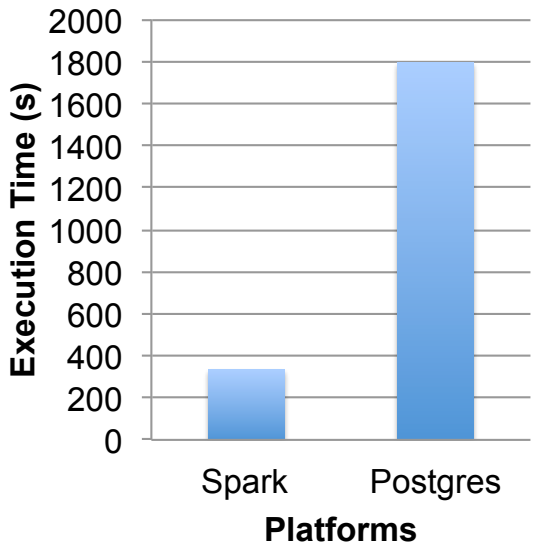

Figure 5.1: Baseline SIRUM on Spark vs. PostgreSQL

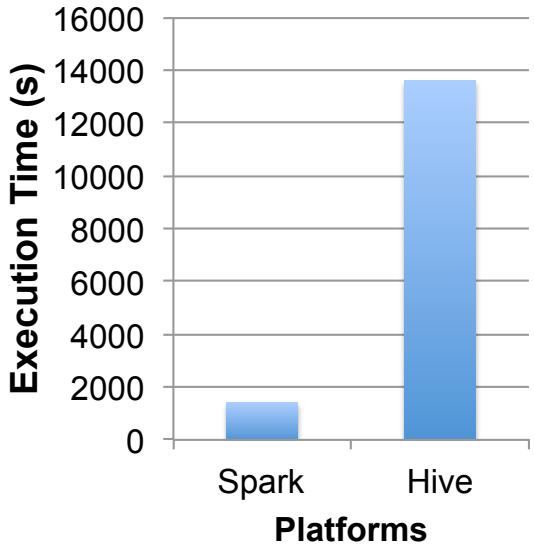

Figure 5.2: Baseline SIRUM on Spark vs. Hive

I/O for saving and retrieving intermediate results. In contrast, Spark caches parts of, if not all, intermediate results as RDDs in the cluster memory. We also observed that launching and cleaning up tasks are slower for SIRUM on Hive.

While we only present the experiment results using $k = 10$ and $|s| = 16$ for sample-based candidate pruning, we observed similar patterns as the value of k and $|s|$ varied, which did not alter the conclusion of our analysis.

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 that we believe 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 plan for future work to evaluate SIRUM on newer versions of SparkSQL.

The experiments from Section 5.3 to Section 5.5 measure the performance improvements of individual optimizations from Chapter 4 compared to Baseline SIRUM (except SIRUM on sample data, which will be evaluated in Section 5.7 in the context of scalability).

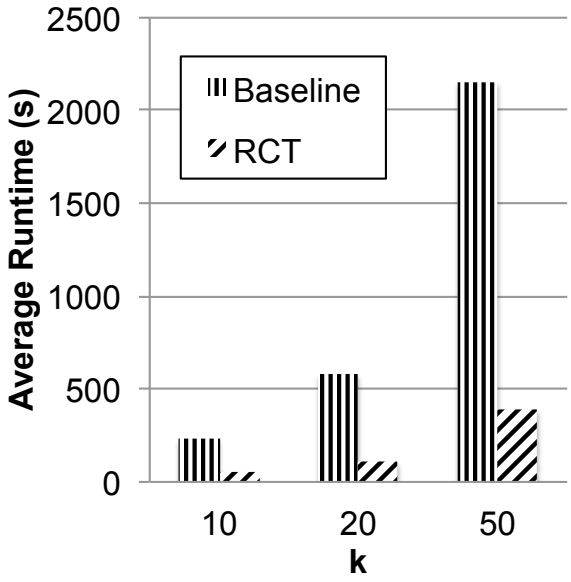

Figure 5.3: Performance improvement of RCT (GDELT)

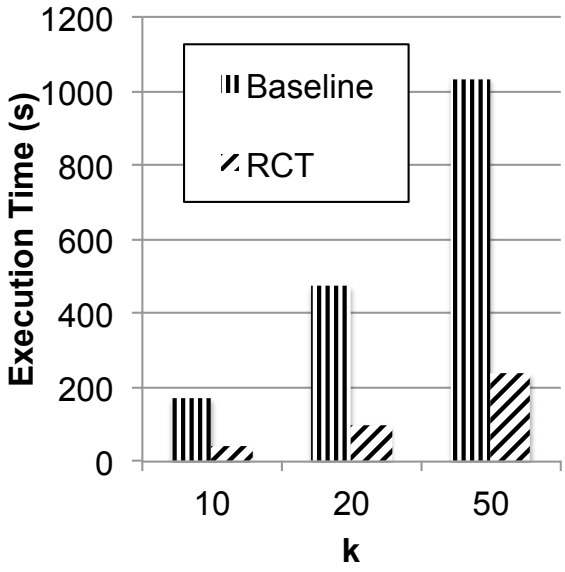

Figure 5.4: Performance improvement of RCT (SUSY)

5.3 Fast Iterative Scaling

We start by comparing the performance of Baseline SIRUM against RCT SIRUM for iterative scaling. Figures 5.3 and 5.4 show the total running time of iterative scaling alone (without rule generation, which is not affected by this optimization) for the GDELT and SUSY datasets, respectively, as we gradually increase the number of rules added to the rule set, k. As shown in both plots, RCT SIRUM is four to five times faster on both datasets for all tested values of k.

5.4 Fast Rule Generation

Next, we evaluate the two optimizations for rule generation: fast candidate pruning and fast candidate rule processing. Figure 5.5 compares the total running time of rule generation alone (without iterative scaling, which is not affected by this optimization) of Baseline SIRUM and SIRUM with FastPruning SIRUM for different values of $|s|$. We used the GDELT dataset with k=20. We observed similar results with other values of k in the range from 10 to 50. FastPruning SIRUM is able to achieve roughly a factor of two speedup, and it appears that the speedup increases as $|s|$ increases.

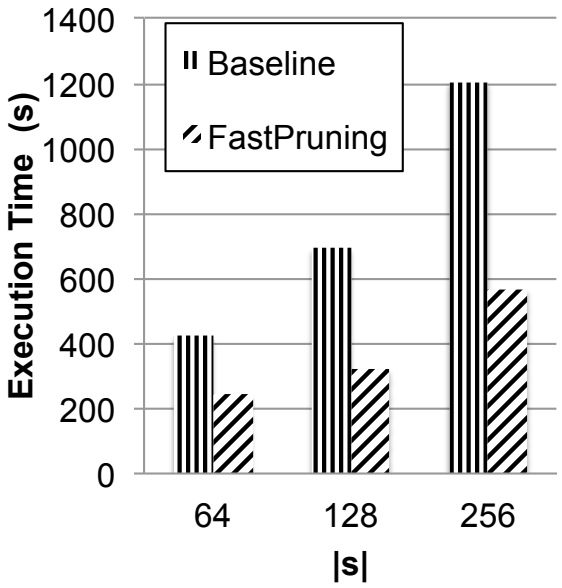

Figure 5.5: Performance improvement of fast candidate pruning (k=20)

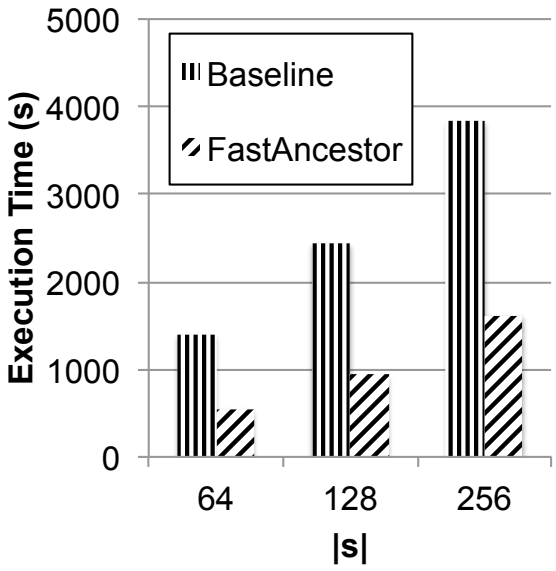

Figure 5.6: Performance improvement of fast rule generation (k = 20)

As shown in Figure 3.2, for the SUSY data set, the running time of rule generation is dominated by ancestor generation, so fast candidate pruning has little impact on the overall running time. Instead, we now evaluate fast candidate rule processing on the SUSY dataset. Figure 5.6 shows the total running time of rule generation for Baseline SIRUM and FastAncestor SIRUM for different values of $|s|$, with k=20. The dimension attributes are partitioned evenly into two groups in this experiment. The running time decreases by a factor of about 2.5.

In Figure 3.2, we tested different projections of SUSY and showed that as the number of attributes increases, ancestor generation becomes the bottleneck in rule generation. In Figure 5.7, we show the total running time of rule generation for Baseline SIRUM and FastAncestor SIRUM for the SUSY datasets projected over 10 to 18 attributes; we set k=10 and $|s|$=64 but other values of these parameters showed similar trends. As the number of dimension attributes increases, so does the magnitude of speedup due to the optimization. This is because fast candidate rule processing reduces the number of ancestors generated as intermediate results during rule selection, as we show in Figure 5.8. Note that the y-axis is logarithmic.

Finally, we remark that increasing the number of column groups (recall Section 4.3) beyond two only delivered a slight performance improvement (no more than 20%). The total number of ancestors generated was smaller, but there was more overhead due to multiple stages of computation in Spark.

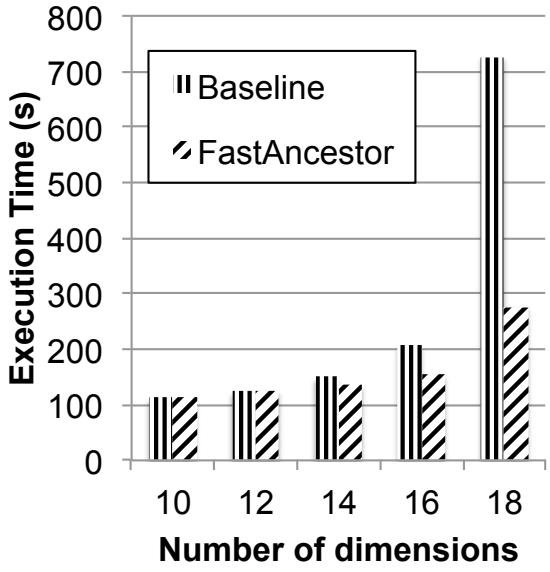

Figure 5.7: Rule generation running times versus the number of dimension attributes.

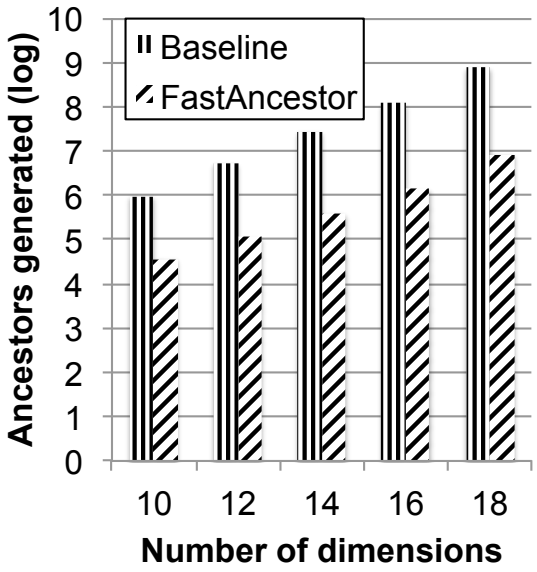

Figure 5.8: Number of ancestors generated versus the number of dimension attributes.

5.5 Adding Multiple Rules per Iteration

We now compare Baseline SIRUM against Multi-rule SIRUM, where l rules instead of one are selected in each iteration, so long as the additional rules are mutually disjoint and lie within the top 1% of rules when sorted by information gain. We test with $l = 2$ and $l = 3$.

Figure 5.9 shows the total running time of rule generation (without iterative scaling) for different values of k using the `GDELT` dataset and $|s| = 256$. We also compare against the running time of "l-rule*", in which Multi-rule SIRUM keeps adding new rules into R until it reaches a commensurate amount of reduction in KL-divergence as Baseline SIRUM; recall that it may require more than k rules, when choosing two or more rules at a time, to obtain the same KL-divergence as Baseline SIRUM which chooses one rule at a time k times. As expected, 2-rule SIRUM reduces the running time of rule generation roughly by about a half (and has little effect on the total running time of iterative scaling since the total number of rules generated is still k). However, to obtain the same KL-divergence as Baseline SIRUM, 2-rule SIRUM needs to generate more than k rules and therefore its performance improvement drops slightly. While 3-rule SIRUM further reduces the running time of rule generation, the improvement over 2-rule SIRUM is marginal: there is the additional overhead of finding the top three non-overlapping rules. Moreover, it takes slightly longer for 3-rule* to achieve the same KL-divergence than

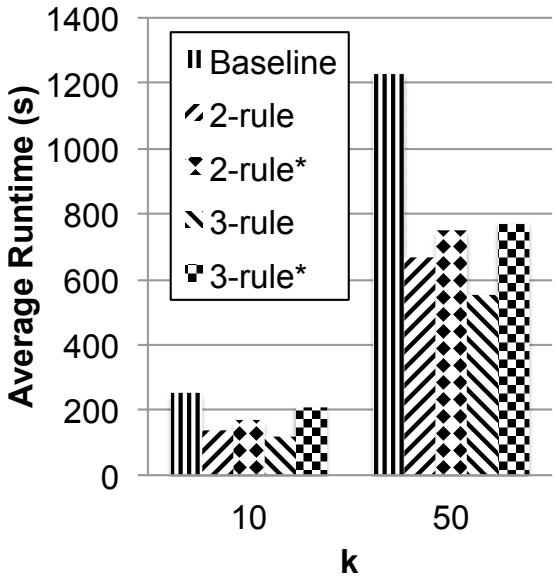

Figure 5.9: Performance of Multi-rule SIRUM (GDELT)

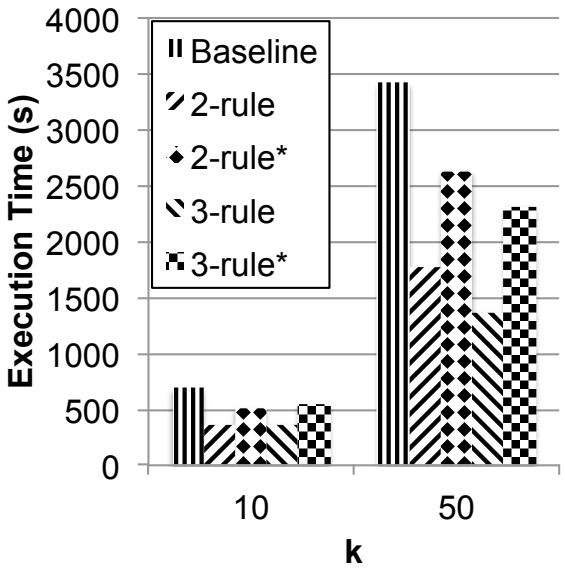

Figure 5.10: Performance of Multi-rule SIRUM (SUSY)

2-rule* (i.e., more rules in total).

The number of additional rules required to match the KL-divergence of a one-rule-at-a-time approach depends on the data. Figure 5.10 shows the corresponding plot for the SUSY dataset. Similar to the result for the GDELT dataset, 2-rule SIRUM reduces the running time roughly in a half. Furthermore, 2-rule* is even slower relative to Multi-rule than in Figure 5.9. That is, 2-rule SIRUM requires even more additional rules: e.g., for $k = 50$, multi-rule SIRUM requires 36 iterations, or up to 72 rules, to obtain the same KL-divergence as Baseline SIRUM which performs 50 iterations and adds one rule at a time. As for 3-rule SIRUM, we observe a similar pattern as in the GDELT dataset. However, it takes less time for 3-rule* to achieve the same KL-Divergence than 2-rule*.

Since 3-rule SIRUM only provides marginal performance benefit to 2-rule SIRUM, we limit the number of rules added per iteration to 2 for any optimizations involving Multi-rule SIRUM in the rest of this chapter.

5.6 Optimized SIRUM vs. Prior Work

In this section, we present experimental results of Optimized SIRUM to evaluate the compound effect of all iterative scaling and rule generation optimizations (but not SIRUM on sample data) on two applications: informative rule mining and data cube exploration. In particular, we compare Optimized SIRUM to straightforward distributed implementations of existing techniques for informative rule mining over binary measure attributes from [16] and smart data cube exploration from [29].

5.6.1 Informative Rule Mining

In this application, the mining algorithm is generating a set of k informative rules assuming a binary measure attribute (and a slightly different definition of KL-divergence for binary attributes), as detailed in [16]. We test the following implementations: Naive SIRUM, Baseline SIRUM (using broadcast join) and Optimized SIRUM (including fast iterative scaling, fast candidate pruning, fast ancestor generation, and selecting two rules at a time). We also test Optimized*, which corresponds to running Optimized SIRUM until it reaches the same KL-divergence as Naive and Baseline SIRUM which choose one rule at a time. Naive SIRUM corresponds to the distributed implementations of the techniques from [16].

We start by showing the scalability of Optimized SIRUM on *TLC_2m* and larger samples of `TLC` up to *TLC_40m*. Figure 5.11 shows the running times for $k = 20$ and $|s| = 64$. Baseline SIRUM already significantly outperforms Naive SIRUM due to the use of broadcast joins, and Optimized SIRUM further improves Baseline SIRUM by a factor of five. Even with one rule inserted per iteration, Optimized SIRUM is still two to three times faster. Moreover, the performance improvement increases as the data size increases.

In the remainder of this section, we omit Naive SIRUM and show the benefits of Optimized SIRUM over Baseline SIRUM.

Next, we present the running time improvement for different values of k over the `GDELT` (Figure 5.12; $|s|$=256) and `SUSY` (Figure 5.13; $|s|$=64) datasets. Observe that Optimized SIRUM is consistently five times faster than the Baseline.

Finally, we examine the performance improvement for different values of $|s|$. Figure 5.14 plots the percentage of performance improvement as a function of $|s|$ for the `Income` and `SUSY` datasets. Optimized SIRUM consistently achieves 80 percent improvement, i.e., factor of five.

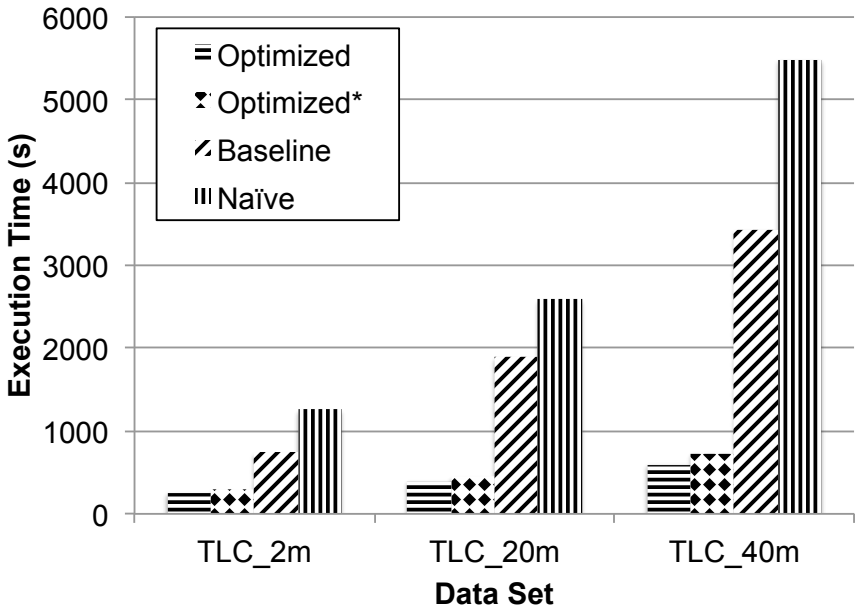

Figure 5.11: Performance improvement on rule mining (TLC)

5.6.2 Data Cube Exploration

In this section we evaluate the performance of Baseline and Optimized SIRUM for data cube exploration [29]. We assume that the user has examined the results of two group-by queries (out of all the possible group-by queries on the dimension attributes) with the lowest cardinality. Then SIRUM performs iterative scaling and present to the user the k most informative rules with respect to what the user has seen so far. We do not use candidate pruning in this experiment, as it was not originally implemented in [29]; thus, here Optimized SIRUM only includes fast iterative scaling, fast ancestor generation and multi-rule selection.

Figure 5.15 shows the running time of data cube exploration implemented as Baseline and Optimized SIRUM, as well as Optimized SIRUM without multi-rule insertion. We use the GDELT dataset and $k = 10$. We also separately show the running time of iterative scaling and rule generation. We observe a factor of 10 performance improvement for Optimized SIRUM and about a factor of six improvement for Optimized SIRUM achieving the same level of information gain as Baseline. The reason why Baseline spends so much time on iterative scaling is that we followed implementation of the iterative scaling algorithm exactly as described in [29], which turned out to be less efficient than Algorithm 1, even before applying the RCT optimization. The main difference is that [29] resets all the λ's to one and re-scales them whenever a new rule is

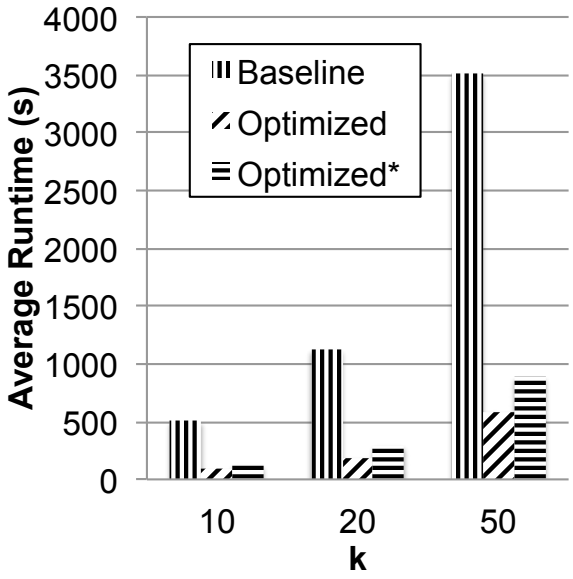

Figure 5.12: Performance improvement on rule mining (GDELT)

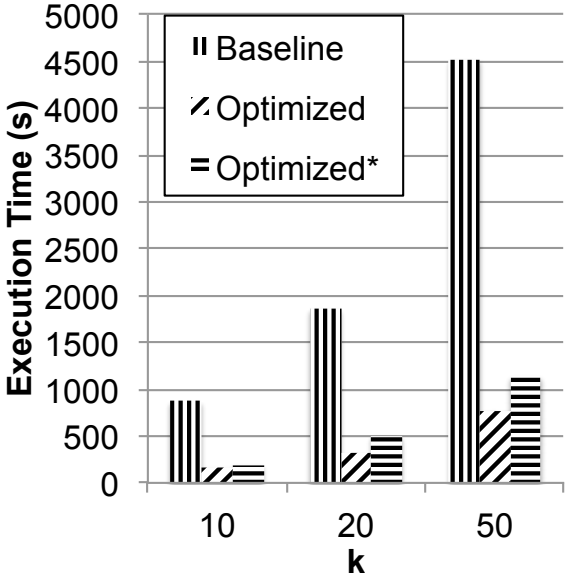

Figure 5.13: Performance improvement on rule mining (SUSY)

added, whereas Algorithm 1 carries over previous λ's as new rules are being added.

5.7 Scalability

5.7.1 Strong Scalability

To measure strong scalability [19] of (Optimized) SIRUM, we keep the data size fixed and vary the number of Spark executors. Figure 5.16 shows the results for *TLC_2m* and the 10 times larger *TLC_20m* with $|s| = 64$ and $k = 10$. For the smaller data set, we observe only a factor of 3 performance improvement as the number of executors increases from 2 to 16. The sub-linear scalability likely means that Spark cannot make full use of additional resources and that the communication overhead increases in proportion to the number of nodes. However, for *TLC_20m*, SIRUM obtains a factor of 6 performance improvement as we scale the cluster 8 times larger. It is important to understand the super-linear scaling effect when the cluster size grows from 2 to 4 executors. Further analysis shows that the size of the working set exceeds the capacity of 2 executors but is less than that of 4 executors. Some of the cached RDD blocks in

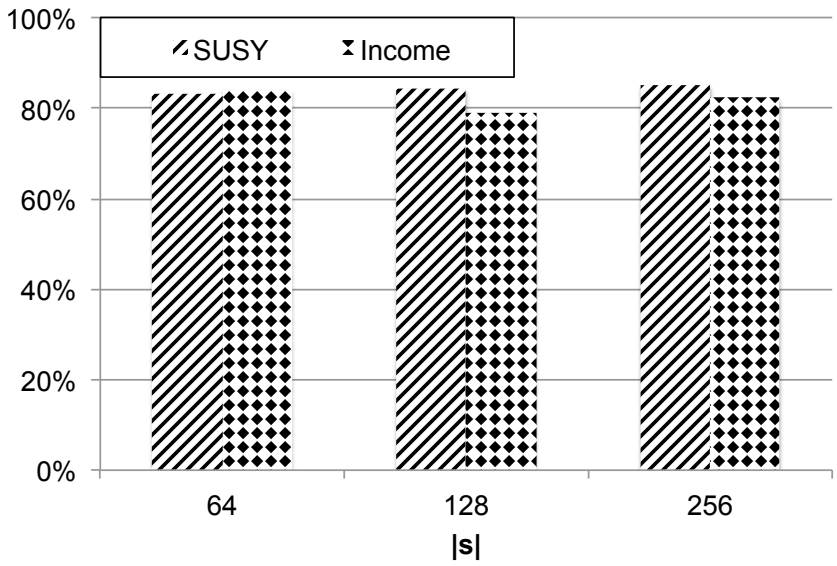

Figure 5.14: Performance improvement on rule mining vs. $|s|$

the working set are evicted from the main memory and have to be recomputed or retrieved from disk afterwards.

5.7.2 Weak Scalability

To measure weak scalability [19], we gradually increase the data size and also proportionally increase the number of executors. Figure 5.17 shows the results. The first data point corresponds to 4 executors and *TLC_40m*, the second data point uses twice the data and twice the number of executors (8 and *TLC_80m*), and the third data point again uses twice as much data and twice as many executors. Ideally, the plot is expected to follow a straight horizontal line, but the actual results show that the running time increases slightly as the data size increases, despite adding more executors. To understand this, we measured the execution time of individual Spark tasks running during the execution of SIRUM, each of which is scheduled on a separate "executor core" in Spark. It turns out the variance is caused by stragglers; the median of task execution time is the same but tasks on certain nodes take noticeably longer to complete. We believe the problem could be mitigated with the help of speculative execution or full cloning of small jobs [5].

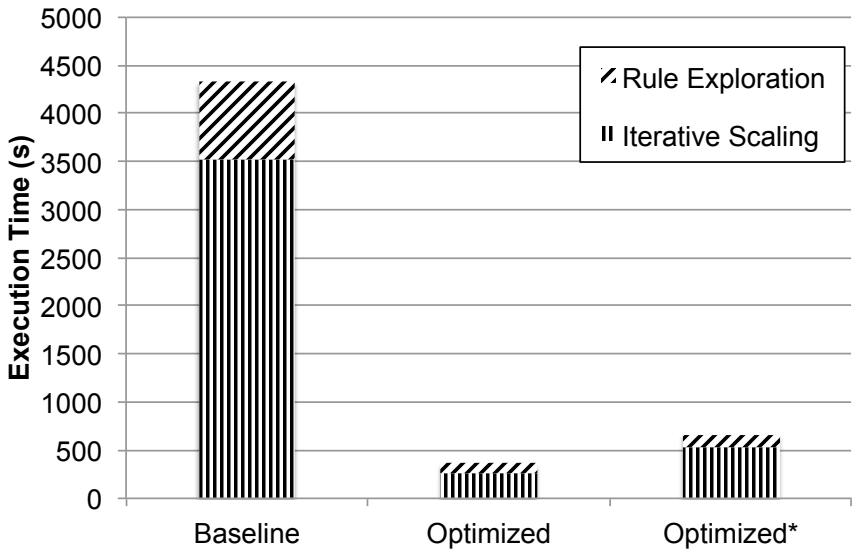

Figure 5.15: SIRUM performance on data cube exploration

5.7.3 SIRUM on Sample Data

In this section, we examine the utility of SIRUM on sample data proposed in Section 4.5. 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, and the KL-divergence penalty from working with a sample is small.

Figures 5.18 and 5.19 illustrate the running time and information gain as we vary the sampling rate from 100 percent down to 10, 1 and 0.1 percent. The former uses `TLC` and 16 X 45 GB executor memory; 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 a significant performance improvement (4X or higher) with a 10 percent sample, which is small enough to be cached in memory. 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.

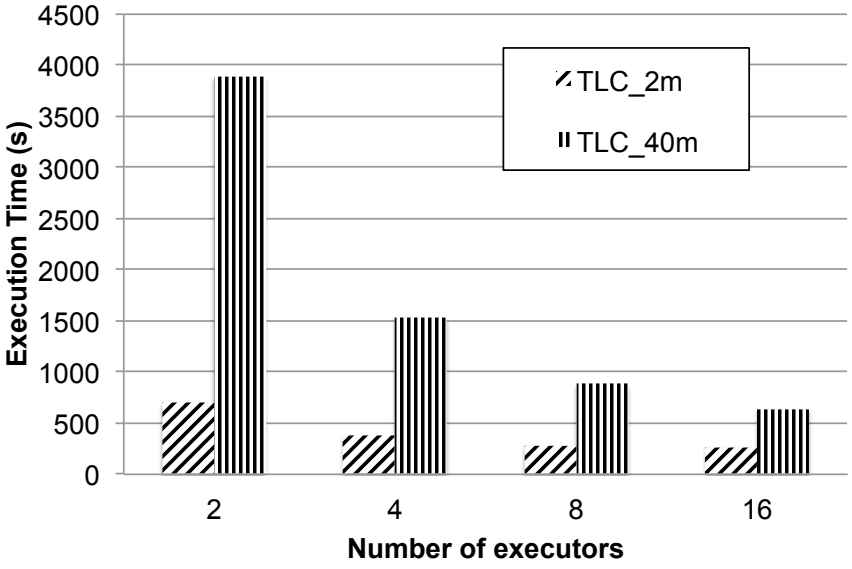

Figure 5.16: Strong scaling of Optimized SIRUM

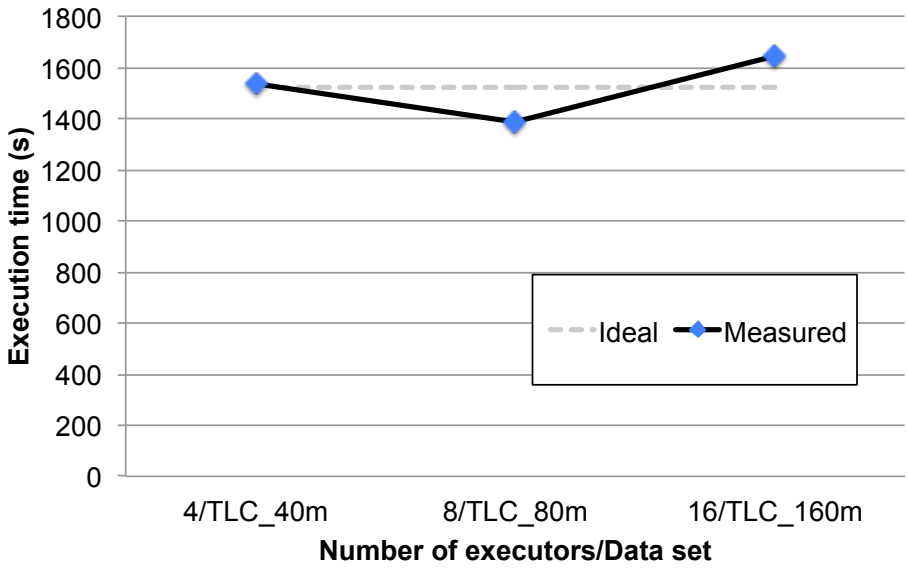

Figure 5.17: Weak scaling of Optimized SIRUM

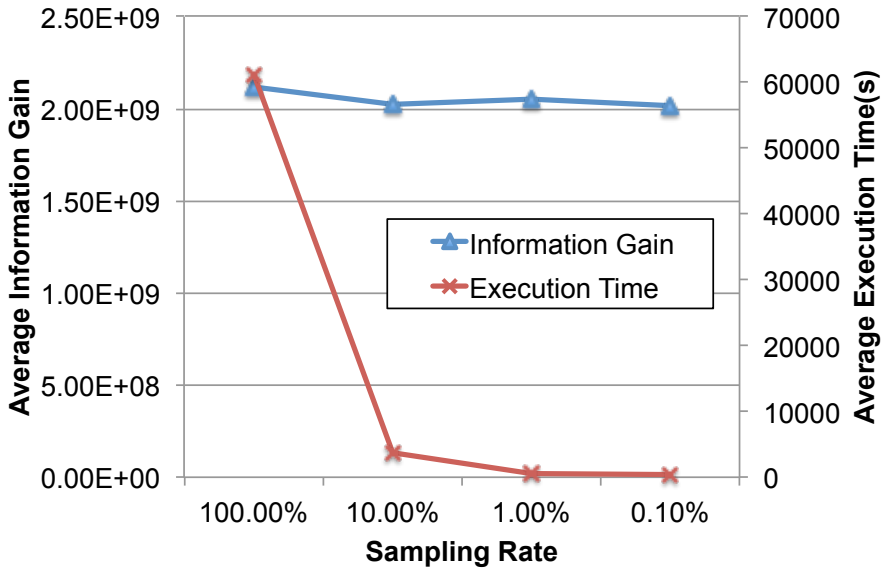

Figure 5.18: Execution time and information gain of SIRUM on sample data over the `TLC` data set (16 X 45 GB executor memory)

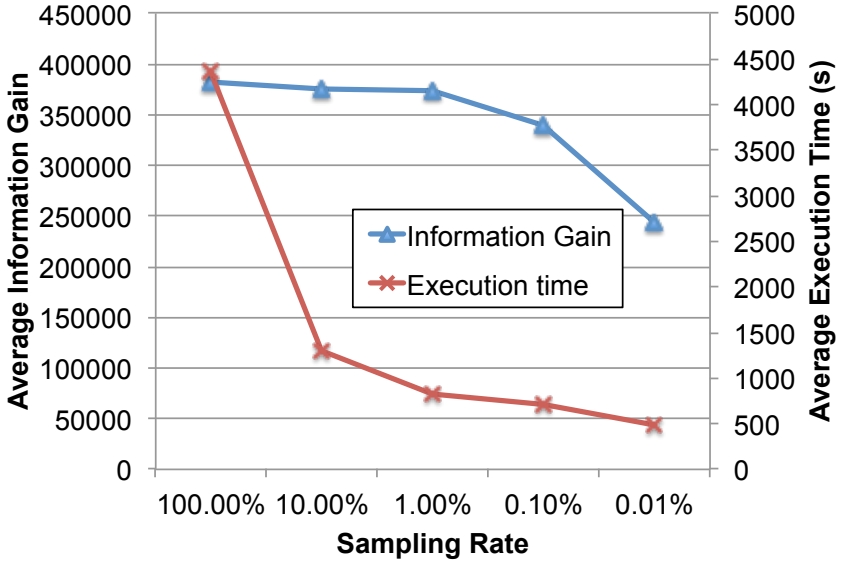

Figure 5.19: Execution time and information gain of SIRUM on sample data over the `SUSY` data set (8 GB executor memory)

Chapter 6

Related Work

Informative rule mining has appeared in several contexts, including data summarization (e.g., of multidimensional data with a binary measure attribute [16] and itemset data [24]) and data cube exploration [29]. Diagnosing data quality problems can also benefit from informative rule mining, although previous work in this area such as Data X-Ray [35] and Data Auditor [17] used different techniques. SIRUM may be used in all of these applications to improve performance and scalability.

Some performance improvements proposed in existing work optimize specific sub-problems (e.g., finding informative itemsets [24]). Others limit the space of candidate rules; e.g., by disallowing overlapping rules unless one is contained within the other [29], or by only considering those which can be generated from a random sample, i.e., sample-based candidate pruning [16]. SIRUM implements and further optimizes candidate pruning, and includes other novel optimizations of informative rule generation and iterative scaling. We know of no other work on distributed informative rule mining.

Focusing on the operations involved in informative rule mining, optimizations of iterative scaling have been proposed in [24], but in the context of itemsets; our optimizations are more general. In terms of rule generation, our fast processing of candidate rules is conceptually similar to optimizations of hash-based data cube algorithms (see, e.g., [3]), in that both re-use previously computed parts of a data cube to compute other parts. However, we are not aware of prior applications of this idea to a map-reduce context. There is recent work on distributed data cube computation that takes stragglers into account [25] and may be added to SIRUM to further improve performance. There has also been work on sort-based rather than hash-based distributed data cube computation [22], but it is not clear how to incorporate sample-based candidate pruning into such an approach (if we sort the data and then prune the space of candidate rules, we

are paying for sorting the entire data set anyway). Furthermore, there is earlier work on parallel data cube computation (see, e.g., [11, 15]), but it is not clear if it can be adapted for Spark or map-reduce. Finally, SIRUM on sample data is akin to other work on sample-based computation over big data such as BlinkDB [2].

Chapter 7

Conclusion

In this thesis we presented SIRUM, a scalable approach to informative rule mining. We proposed several optimizations of the rule mining process and implemented them in the Spark distributed processing system. Based on experiments on real data, we showed that an improved design of SIRUM is more than an order of magnitude faster than a naive approach corresponding to prior work, and five times faster than a baseline approach that incorporates straightforward improvements such as broadcast joins.

As for future work, we are currently investigating several options to eliminate redundancy in ancestor rule generation; if a rule has the same support set as one of its descendants, it is unnecessary to evaluate it because its gain is the same as its descendant's. In addition, it should be possible to optimize the data cube operations in candidate rule generation (and also optimize data cube computation in Spark). We are also interested in building a streaming version of SIRUM (e.g., using Spark Streaming) that incrementally maintains informative rules as new data arrive. Moreover, we are exploring possible extensions to the informative rule mining problem. Specifically, it might be interesting to study the correlation among multiple measure attributes as a function of the dimension attributes. Finally, we recognize that informative rule mining over high-dimensional datasets is a challenging problem. Even with sample-based candidate pruning, the number of candidate rules still grows exponentially with respect to the number of dimension attributes.

References

[1] Spark programming guide. http://spark.apache.org/docs/latest/programming-guide.html. Accessed: 2015-07-31.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[3] Sameet Agarwal, Rakesh Agrawal, Prasad M Deshpande, Ashish Gupta, Jeffrey F Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB*, volume 96, pages 506–521, 1996.

[4] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, 1993.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI*, volume 13, pages 185–198, 2013.

[6] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5, 2014.

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

[8] Spyros Blanas, Jignesh M Patel, Vuk Ercegovac, Jun Rao, Eugene J Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 975–986. ACM, 2010.

[9] Alain Casali, Rosine Cicchetti, and Lotfi Lakhal. Cube lattices: A framework for multidimensional data mining. SIAM.

[10] Songting Chen. Cheetah: a high performance, custom data warehouse on top of mapreduce. *Proceedings of the VLDB Endowment*, 3(1-2):1459–1468, 2010.

[11] Ying Chen, Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Parallel ROLAP data cube construction on shared-nothing multiprocessors. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–pp. IEEE, 2003.

[12] Imre Csiszár. I-divergence geometry of probability distributions and minimization problems. *The Annals of Probability*, pages 146–158, 1975.

[13] John N Darroch and Douglas Ratcliff. Generalized iterative scaling for log-linear models. *The annals of mathematical statistics*, pages 1470–1480, 1972.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[15] Frank Dehne, Todd Eavis, and Andrew Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 319–326. Springer, 2001.

[16] Kareem El Gebaly, Parag Agrawal, Lukasz Golab, Flip Korn, and Divesh Srivastava. Interpretable and informative explanations of outcomes. *Proceedings of the VLDB Endowment*, 8(1):61–72, 2014.

[17] Lukasz Golab, Howard Karloff, Flip Korn, and Divesh Srivastava. Data auditor: Exploring data quality and semantics using pattern tableaux. *Proceedings of the VLDB Endowment*, 3(1-2):1641–1644, 2010.

[18] Edwin T Jaynes. Information theory and statistical mechanics. *Physical review*, 106(4):620, 1957.

[19] Alan Kaminsky. Big cpu, big data: Solving the world's toughest computational problems with parallel computing. *Creative Commons*, 2013.

[20] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.", 2015.

[21] Solomon Kullback. *Information theory and statistics*. Courier Corporation, 1968.

[22] Suan Lee, Jinho Kim, Yang-Sae Moon, and Wookey Lee. *Efficient distributed parallel top-down computation of ROLAP data cube using mapreduce*. Springer, 2012.

[23] Kalev Leetaru and Philip A Schrodt. GDELT: Global data on events, location, and tone, 1979–2012. In *ISA Annual Convention*, volume 2, page 4, 2013.

[24] Michael Mampaey, Nikolaj Tatti, and Jilles Vreeken. Tell me what i need to know: succinctly summarizing data with itemsets. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 573–581. ACM, 2011.

[25] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. Data cube materialization and mining over mapreduce. *Knowledge and Data Engineering, IEEE Transactions on*, 24(10):1747–1759, 2012.

[26] Regina O Obe and Leo S Hsu. *PostgreSQL: Up and Running: A Practical Introduction to the Advanced Open Source Database*. " O'Reilly Media, Inc.", 2014.

[27] B Robert. Ash. information theory, 1990.

[28] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1357–1369. ACM, 2015.

[29] Sunita Sarawagi. User-cognizant multidimensional analysis. *The VLDB journal*, 10(2-3):224–239, 2001.

[30] Murat Sariyar, Andreas Borg, and Klaus Pommerening. Controlling false match rates in record linkage using extreme value theory. *Journal of biomedical informatics*, 44(4):648–654, 2011.

[31] Philip A Schrodt, Omür Yilmaz, Deborah J Gerner, and Dennis Hermreck. The cameo (conflict and mediation event observations) actor coding framework. In *2008 Annual Meeting of the International Studies Association*, 2008.

[32] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[33] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[34] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al.

Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[35] Xiaolan Wang, Xin Luna Dong, and Alexandra Meliou. Data X-Ray: A diagnostic tool for data errors. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1231–1245. ACM, 2015.

[36] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[37] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[38] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

APPENDICES

Appendix A

Correctness of Column Grouping

To examine the correctness of the column grouping optimization, it is necessary to show that it satisfies the following properties. First, FastAncestor SIRUM should generate the same set of ancestor rules from $LCA(s, D)$ as Baseline SIRUM. Second, the aggregate values must remain the same for each ancestor rule generated. The two properties together guarantee that SIRUM is able to evaluate the same set of candidate rules with correct information gains after the column grouping optimization is applied.

Theorem 1. *Let X and Y be two sets of candidate rules generated from the same set of LCA rules $LCA(s, D)$ by FastAncestor SIRUM and Baseline SIRUM respectively, we have $X = Y$. In addition, the aggregate values associated with the candidate rules are also the same.*

Proof. We first prove that the $X = Y$. It is trivial to see that $LCA(s, D) \subseteq X$ and $LCA(s, D) \subseteq Y$. We only need to show (1) for any generated ancestor rule $r \in X$, $r \in Y$; and (2) for any generated ancestor rule $r \in Y$, $r \in X$.

Suppose that FastAncestor SIRUM partitions the dimension attributes into n column groups and process them following the order G_1, G_2, ..., G_n. The set of ancestors generated after processing G_i is denoted by Q_i. For consistency we denote $LCA(s, D)$ by Q_0. It follows that $X = \bigcup_{i=0}^{n} Q_i$.

1. Let $r \in X$. Since $r \notin LCA(s, D)$, there exists at least one Q_l such that $r \in Q_l$ where $l > 0$. Because rules in Q_l are the ancestor rules generated from descendant rules in Q_{l-1} by replacing one or more non-wildcard values for attributes included in G_l, there exists at least one descendant rule of r in Q_{l-1} that generates r; otherwise, $r \in Q_{l-1}$ if all values of

attributes in G_l are wildcards. Through induction, we can show that there exists an LCA rule in $r' \in Q_0$ such that r is either an ancestor of r' or $r' = r$. Note that Y includes all ancestor rules generated from rules in $LCA(s, D)$. Hence, $r \in Y$.

2. Let $r \in Y$. By definition, for any $r \in Y$ there exists at least one $r_0 \in LCA(s, D)$ such that r is an ancestor generated from r_0 by Baseline SIRUM. Let $E = \{l_1, l_2, \ldots, l_w\}$ be the set of attributes such that $p[A_i] = r_0[A_i]$ for $i \notin E$ whereas $r[A_i] =$ ' $*$ ' and $p[A_i] \neq$ ' $*$ ' for $i \in E$. That is, r can be generated by replacing all attributes values of A_i with wildcards where $i \in E$. After G_1 is processed, Q_1 includes all possible ancestor rules generated by replacing zero or more non-wildcard values for attributes in G_1. It follows that there exists $r_1 \in Q_1$ such that r_1 is generated by replacing p_0's non-wildcard values for attributes in $E \cap G_1$. Similarly, there exists $r_2 \in Q_2$ such that r_2 is generated by replacing r_1's non-wildcard values for attributes in $E \cap G_1$. After G_n is processed, the r_n is generated by replacing non-wildcard values for attributes in $\bigcup_{i=1}^{n}(E \cap G_i)$. Since $E \subseteq \{A_1, A_2, \ldots, A_d\} = \bigcup_{i=1}^{n} G_i$, we have $E = \bigcup_{i=1}^{n}(E \cap G_i)$. It follows that $r = r_n$. Therefore, $r \in X$.

Given that FastAncestor SIRUM generates the same set of ancestor rules from $LCA(s, D)$ as Baseline SIRUM, the aggregate values also remain the same for each ancestor rule as long as an LCA rule generates only one *instance* for each of its ancestor rules. Let l'_1, l'_2, ..., l'_w be a permutation of $E = \{l_1, l_2, \ldots, l_w\}$. An instance of an ancestor rule r is created when $r_0[A_{l'_1}]$, $r_0[A_{l'_2}]$, ..., $r_0[A_{l'_w}]$ are replaced by wildcards in that order. Two instances of the same ancestor are generated by an LCA rule only if there exist two distinct permutations of elements in E. Note that the processing order G_1, G_2, ..., G_n defines a partial order over $l \in E$. Within each group, FastAncestor SIRUM follows a processing order based on the subscript value. For example, $r_0[A_{l'_i}]$ is replaced with a wildcard before $r_0[A_{l'_j}]$ if and only if $l'_i < l'_j$. It follows that a permutation is uniquely defined for each pair of r and r_0. Therefore, an LCA rule generates only one instance for each of its ancestor rules under FastAncestor SIRUM. □