Parallel Feature Selection Using Only Counts

Staal A. Vinterbo* Jialan Que†

Abstract

Count queries belong to a class of summary statistics routinely used in basket analysis, inventory tracking, and study cohort finding. In this article, we demonstrate how it is possible to use simple count queries for parallelizing sequential data mining algorithms. Specifically, we parallelize a published algorithm for finding minimum sets of discriminating features and demonstrate that the parallel speedup is close to the expected optimum.

1 Introduction

A fundamental problem in data analysis, data mining, and machine learning is feature selection. A particular instance of this problem is finding a minimum set of data attributes that preserve a given equivalence relation in the data. Examples of this instance for varying equivalence relations include finding minimal length candidate keys in databases [4], finding minimal length classification rule antecedents [10], and haplotype tagging by minimal sets of single nucleotide polymorphisms [15]. The general problem is well known to be NP-hard and polynomial time approximation properties can be related to those of set covering [15].

The notion of polynomial time being efficient is problematic when quantities become really large, such as in next generation sequencing experiments where data from single experiments is measured in terabytes [12]. Even low exponent polynomial time algorithms can be impractical under such circumstances.

Our main technical contribution is a formulation of a previously published efficient sequential approximation algorithm for attribute selection [15] in terms of

*Staal.Vinterbo@ntnu.no, Norwegian University of Science and Technology. This work was funded in part by NIH NLM grant 7R01LM007273-07 and NIH Roadmap for Medical Research grant U54 HL108460. We thank Guilherme Schafer for help with setting up the hardware for the experiments.

†jialan.que@gmail.com, Fair Isaac Corporation.

This paper was presented at the NIK-2018 conference; see http://www.nik.no/.

count queries. We demonstrate that this formulation achieves a parallel speedup close to the expected optimum when data access is distributed. Furthermore, our experiments suggest that count queries can serve as a data access mechanism that allows efficient parallelization of sequential algorithms that can be difficult to achieve otherwise.

2 Related work

Count queries belong to a class known as online analytical processing (OLAP) queries, often considered in the context of business intelligence and statistical databases. In public health, count queries are essential operations for most automatic biosurveillance systems where the time-series data which these systems analyze are responses to count queries [11,13]. In general, quantitative queries, i.e., queries that return either reals or integers, have been used to analyze combinatorial problems such as graph reconstruction [6], finding counterfeit coins using scales [7], learning concept classes in the context of statistical learning theory [9], and afford private data access [5].

Counts are also used to compute small synopses or “sketches” of large amounts of data that can subsequently be used for efficiently approximating a class of queries [3]. However, the class of queries that can be approximated this way is limited [8].

Our work also relates to Blelloc et al.'s work on parallel set-cover [2], except that their approach does not preserve established non-parallel approximation bounds of solution optimality.

3 Minimum sets of discerning attributes

Any partition of a set X has an equivalence relation associated with it, consisting of pairs of elements that are co-located in the same equivalence class. The complement of this equivalence relation consists of all pairs of elements found in different classes. For attribute set induced partitions, this means that these pairs are *discernible* by the set of attributes that induced the partition. Formally, let

$$\pi(a) = \{(x, y) \in X^2 | a(x) \neq a(y)\}$$

denote the set of pairs of elements in X that are discernible by attribute a. For a subset $A' \subseteq A$ of attributes A on X, we let the pairs discernible by A' be $\pi(A') = \cup_{a \in A'} \pi(a)$.

Given a function f on X, we can express our general problem as wanting to find a minimum A' such that $\pi(A') \cap \pi(f) \supseteq \pi(A) \cap \pi(f)$. The set $M(A, f)$ defined as

$$M(A, f) = \{B \subseteq A | \pi(B) \cap \pi(f) \supseteq \pi(A) \cap \pi(f)\}$$

is the set of candidate solutions among which we seek a minimum cardinality element. Our previous analysis [14, 15] used the following simple but effective greedy algorithm for finding the smallest element of $M(A, A)$: Add to the initially empty solution S the attribute a for which $\pi(a)$ contains the most elements of $\pi(A)$ not already contained in $\pi(S)$. Continue doing this until $\pi(S) \supseteq \pi(A)$. This is essentially a set-covering based approximation algorithm.

The attribute chosen in the simple algorithm above is the attribute a that maximizes $|(\pi(A) - \pi(S)) \cap \pi(a)|$. We now note that for any set $\Pi \subseteq X^2$, maximizing the expression $|(\Pi - \pi(S)) \cap \pi(a)|$ is equivalent to minimizing $|\Pi - \pi(S \cup \{a\})|$. This latter quantity $|\Pi - \pi(S \cup \{a\})|$ is the number of left over pairs in Π not discerned by $S \cup \{a\}$. We can express this quantity as follows:

$$|\Pi - \pi(S \cup \{a\})| = \texttt{leftover}(\Pi, S \cup \{a\}), \tag{1}$$

where `leftover` is defined as:

$$\texttt{leftover}(\Pi, S) = \sum_{x \in X} \frac{|\Pi \cap ([x]_S)^2|}{|[x]_S|}.$$

Algorithm formulations

The `mda` function in (2) implements the greedy MDA algorithm outlined above. The inputs are Π – a set of pairs, A – a set of attribute functions, and a real value τ in the unit interval. The algorithm minimizes (1), and stops once the selected attributes in S discern between a fraction τ of the pairs in Π.

$$\begin{aligned}
&\texttt{mda}(\Pi, \tau, \emptyset, S) = S \\
&\texttt{mda}(\Pi, \tau, A, S) = \textbf{if } \left(\frac{|\pi(S)|}{|\Pi|} \geq \tau\right) S \textbf{ else} \\
&\quad \textbf{let } a = \arg\min_{a \in A} \texttt{leftover}(\Pi, S \cup \{a\}) \\
&\quad \textbf{in } \texttt{mda}(\Pi, \tau, A - \{a\}, S \cup \{a\}) \\
&\texttt{mda}(\Pi, \tau, A) = \texttt{mda}(\Pi, \tau, A, \emptyset)
\end{aligned} \tag{2}$$

If we let $\Pi = \pi(A)$ and $\tau = 1$, we have the exact same algorithm as we used for the haplotype tagging problem, and approximation properties established previously [15] hold. We can however choose to let Π be the set of pairs of elements having a different outcome or label, say $\Pi = \pi(f)$ where f is a labelling function. If used in this way, the solution sets computed can be used as templates for classification rule antecedents such as we have done constructing among others Fuzzy classifiers [16]. The parameter τ then becomes a “noise cancelling” parameter [14]. Note that f could discern between elements that A does not, i.e,

$\pi(f) - \pi(A)$ might be non-empty, and in order to interpret τ in a better "calibrated" manner it can be useful to let $\Pi = \pi(f) \cap \pi(A)$. Otherwise a discerning fraction of 1.0 might never be possible.

We now turn to the formulation of the above algorithm in terms of count queries. Let $\texttt{partition}(a, l)$ be a function that computes the partition of the list l of elements from X induced by a and returns it as a list of equivalence classes. Since we want to iteratively refine a given partition by partitioning each current equivalence class by the same given attribute, we define a function `refine` as:

$$\texttt{refine}(a, L) = \texttt{cat}([\texttt{partition}(a, l) | l \in L]), \tag{3}$$

where `cat` concatenates the lists in its argument. Let $\texttt{hist}(a, l)$ be a function that computes a histogram over values that function a takes for the elements in list l. Given the histogram $(n_1, n_2, \ldots, n_k) = \texttt{hist}(a, l)$, the number of pairs of elements in l that are discerned by a can be expressed as:

$$\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} n_i n_j.$$

This sum can be computed in linear time in the length of the histogram by

$$\begin{aligned}
&\texttt{pairs}([], x, y) = x \\
&\texttt{pairs}(h, x, y) = \\
&\quad \textbf{let } n = \texttt{head}(h) \\
&\quad \textbf{in } \texttt{pairs}(\texttt{tail}(h), x + yn, y + n) \\
&\texttt{pairs}(h) = \texttt{pairs}(h, 0, 0).
\end{aligned}$$

Note that the function `head` returns the first element in the argument list, while the function `tail` returns a list equal to argument list with the head removed.

If we let $\mathcal{C} = [C_1, C_2, \ldots, C_l]$ be a list of equivalence classes induced by S, each represented as a list of elements, the number of pairs discerned by f that are not discerned by S, i.e., $\texttt{leftover}(\pi(f), S)$, can be computed by

$$\texttt{leftover}(f, \mathcal{C}) = \texttt{sum}([\texttt{pairs}(\texttt{hist}(f, C)) | C \in \mathcal{C}])$$

where $\texttt{sum}(l)$ is a function that returns the sum of the elements in the list l.

We are now ready to formulate the MDA algorithm we presented in (2) in terms of `partition` and `hist` as (4). The early halting noise reduction parameter

b is computed as $b = \lceil \tau * |\pi(f)| \rceil$, where τ is the fraction of pairs we want covered.

$$\begin{array}{l} \texttt{mda}(f, b, \mathcal{C}, A, S) = \\ \quad \textbf{if } (\texttt{leftover}(f, \mathcal{C}) \leq b) \textbf{ else} \\ \quad\quad \textbf{let } a = \arg\min_{a \in \mathcal{A}}(\texttt{leftover}(f, \texttt{refine}(a, \mathcal{C}))) \\ \quad\quad \textbf{in } \texttt{mda}(f, b, \texttt{refine}(a, \mathcal{C}), A - \{a\}, S \cup \{a\}) \\ \texttt{mda}(f, \tau, A) = \\ \quad \texttt{mda}(f, \lceil \tau * \texttt{pairs}(\texttt{hist}(f, X)) \rceil, [X], A, []) \end{array} \tag{4}$$

Partitions, histograms, and count queries

In the algorithm in (4), the only actual access to data is made in the functions `partition` and `hist`. Let the function `count` take as input a list of attribute – value pairs, compute a counting query, send it to the data base containing X and return the returned count. Also, again assume we can look up the range of any attribute f using the function `range`.

The function `partition` is very similar to `hist` and both can be expressed in terms of a function `access`

$$\begin{array}{l} \texttt{access}(\texttt{fun}, f, p) = \\ \quad \textbf{let } l = [(f, v) | v \in \texttt{range}(f)] \\ \quad \textbf{in } \texttt{fun}(\texttt{count}, [p + [x] | x \in l]) \end{array}$$

as

$$\begin{array}{rcl} \texttt{hist}(f, p) & = & \texttt{access}(\texttt{map}, f, p) \\ \texttt{partition}(f, p) & = & \texttt{access}(\texttt{filter}, f, p) \\ \texttt{map}(f, l) & = & [f(x) | x \in l] \\ \texttt{filter}(f, l) & = & [x | x \in l, f(x) \neq 0]. \end{array}$$

Counting counts

Let again $n = |X|$ and $m = |A|$. The number of calls to `count` made by a call to $\texttt{hist}(f, p)$ or $\texttt{partition}(f, p)$ is as before exactly $|\text{range}(f)|$, which is $O(n)$. If we assume that all ranges are constant, the cost of `hist` and `partition` is also constant. In either case, let this number of `count` calls be denoted by $c(n)$.

A clever implementation of the MDA algorithm in (4) only needs to compute the **let** $a = \arg\min...$ statment, and can remember the refinement of $\mathcal{C}$ by a computed there, as well as the number of undiscerned pairs of f used in the stopping criterion. In order to analyze the number of calls to `hist` and `partition` made it is useful to note that `mda` essentially does recursive partitioning where

each level of the split tree is split on the same attribute. We can think of this as a proper $c(n)$-ary tree. As the leaves of this tree represent a partition of X, an upper bound on the number of leaves is n. The maximal number of internal nodes for a given number of leaves we find in a binary tree, and is $n-1$ for n leaves. It is at these internal nodes that we perform computations. Also note that the number of levels in this tree corresponds to the size of the solution returned. Since this cannot exceed the number of attributes m we can at most have p internal nodes where $p \leq \min(n-1, m)$. At each internal node of this tree we have to evaluate the remaining attributes to determine which one to split on. For an internal node at level i we have to evaluate $m-i$ attributes. Let n_i be interior node $i \in \{1, 2, \ldots, p\}$, and let $l(n_i)$ denote the level node n_i is on. If e denotes the the number of evaluations we have to make in total, we have that:

$$e \leq \sum_{i=1}^{p}(m - l(n_i)) = pm - \sum_{i=1}^{p} l(n_i) \in O(pm).$$

A more precise bound can be found by considering an upper bound for $\sum_{i=1}^{p} l(n_i)$. We know that one interior node has to be the root, i.e., it is on level 1, all the $p-1$ other have a level at least 2. This yields

$$e \leq pm - (1 + 2(p-1)) = pm - 2p + 1 = p(m-2) + 1.$$

Each of these evaluations require a call to `partition` yielding a list of lenght $c(n)$ for each of which a call to `hist` is made, yielding a `count` count of at most on the order of $c(n)^2$. In total we get that `mda` makes at most on the order of

$$c(n)^2(p(m-2)+1)$$

calls to `count`.

4 Distributed Count Queries

As we can see from the analysis above, the number of calls to `count` that `mda` makes in this case is $O(m(m-2))$ if we consider $c(n)$ constant. We now observe that if we partition X into k parts $X_1, X_2, \ldots, X_k$ and store each X_i in data base d_i, we have that

$$\texttt{count}(p) = \sum_{i=1}^{k} \texttt{countd}(p, d_i). \tag{5}$$

where the function $\texttt{countd}(p, d)$ is a version of `count` that in addition to the pairlist p, takes one extra parameter d, the data base which to query. We denote the parallel version of `mda` for k databases as $\texttt{pmda}_k$.

The function $\texttt{countd}(p, d)$ can be viewed as a remote procedure call. In this setting the parameter d serves as the address to where the query p is sent. Remotely at location d, the local store d executes and returns the result of

$$\texttt{response}(p) = \texttt{process}(p).$$

The `process` function is defined by:

$$\texttt{process}(p) = \texttt{postprocess}(\texttt{performquery}(\texttt{translate}(p))).$$

The function `translate` translates the incoming query representation p to fit the local schema. Doing this translation locally means that the central process does not need to know if the data is spread among disparate schemas. The translated query is passed to `performquery` to produce the real count on the local data store. This real count is then potentially transformed by `postprocess`. One example of such post processing is adding noise to the count in order to enhance privacy protection. In the simplest case, both `translate` and `postprocess` do essentially nothing but passing their arguments along.

We now make the assumption that a local data store can process a count query in time that is linear in the size of the store. If we now partition the data into k equally sized parts, each approximately of size n/k, the computation of the individual stores' counts is $O(n/k)$. However, we need to let these k stores know that we want counts and we need to receive and sum these k counts. One simple way to do this, reflecting (5), would be for `mda` to call $\texttt{count}(p, [d_1, d_2, \ldots, d_k])$ instead of $\texttt{count}(p)$ where this two-argument `count` is defined as

$$\texttt{count}(p, l) = \texttt{sum}(\texttt{pmap}(d \to \texttt{countd}(p, d), l))$$

where $\texttt{pmap}(f, l)$ is a function that works just like `map` but applies the function f to all elements in l in parallel.

Assume that sending queries and receiving counts takes constant time, having `mda` call `count` with the full list of data stores achieves a $O(n/k + k)$ time complexity. The additional k comes from sending, receiving, and computing the sum of the k counts received in responses to the queries.

Organizing the k data stores into a tree of depth on the order of $\log(k)$ and requiring that each store can do a little extra work, we can do better. To this end we now add the extra work into `response`. In our implementation each data store knows the roots of its subtrees, aka. its children. We assume that there exists a list s of these children available for use in `response`. We can now define `response` as

$$\begin{aligned}&\texttt{response}(p) = \\ &\quad \textbf{let } c = [p \to \texttt{countd}(p, d_i) | d_i \in s] \\ &\quad \textbf{in } \texttt{sum}(\texttt{pmap}([p \to \texttt{process}(p)] + c)).\end{aligned}$$

What now happens is that on receipt of a remote $\texttt{countd}(p, d_i)$ call, the store d_i computes in parallel the local count as well querying all its children for theirs. Finally, `response` returns the sum of the results. If we now change the function `mda` to call `count` with the list l consisting solely of the root of our stores tree, we achieve a time complexity of $O(n/k + \log k)$.

In practice, if the communications time is significantly less than the query computation time, we can expect to improve the complexity to approach $O(n/k)$ using a suitable organization of the stores.

Algorithm time complexity

The algorithm we presented for the haplotype tagging problem had running time $O(\frac{np}{2}(2m-p+1))$ where $p = \min(n, m)$. Assuming $m \leq n$ this becomes $O(\frac{nm}{2}(m+1))$ which is $O(nm^2)$. We have established that `mda` issues $O(m(m-2)+1)$ calls to `count` which is $O(m^2)$. Applying the hierarchical structure of partitioned data described above, we achieve a running time for the algorithm of $O(m^2(n/k+\log k))$ for n data points equally partitioned among k stores.

5 Experimental Results

We implemented the parallel algorithm $\texttt{pmda}_k$ described in Section 4 which runs k parallel data stores in k processes. The main $\texttt{pmda}_k$ process queries the k data stores in parallel using the mechanism described in Section 4. We selected three different data sets and compared the running times of $\texttt{pmda}_k$ where $k \in \{2, 4, 8\}$.

Setup

All experiments were run using a Dell PowerEdge M610 with two Intel Xeon E5620 2.393 Ghz 4 core CPUs and 90GB RAM running Ubuntu under VMWare ESXi 5.

Data sets

We selected three data sets from UCI machine learning data repository as initial datasets. The two criteria used to select experimental data sets are 1) the number of the instances should be large enough to distinguish the query time among different strategies and 2) the attributes should be categorical and able to discern most instances to suit the use case of the proposed algorithm. The properties of the three data sets are listed in Table 1.

The first data set was derived from the USCensus 1990 raw data set which includes approximately 2.5 million records. We arbitrarily chose 8 categorical attributes from 68 total attributes and each of them has a number of values ranging from 2 to 4. The second data set is the one used for the Third International Knowledge Discovery and Data Mining Tools Competition. This data set contains

Data set	# of attrs	# of records	sizes of ranges
US Census	8	2,458,285	2-4
KDD Cup 99	10	4,000,000	2-3
Covertype	12	581,012	3-7

Table 1: Description of the three experimental data sets

raw TCP dump data for a local-area network (LAN) simulating a typical U.S. Air Force LAN, which includes a wide variety of intrusions simulated in a military network environment. The original data set has 4 million records and 42 attributes. We selected all 7 attributes which have discrete values and the sizes of the ranges is between 2 and 3. We arbitrarily selected 3 other attributes and converted them into binary ones. The third data set is the covertype data set used to predict forest cover type from cartographic variables. It has approximately a half million instances and 54 attributes in total. We discretized the ten quantitative attributes into three categories each and combined four binary attributes into one. In addition to the classification attribute which has seven values, we converted the data set into one with 12 categorical attributes having the range sizes in $\{3, 4, 7\}$. In order to create datasets of sizes up to 2^{22} records, random subsampling with replacement was used.

Implementation

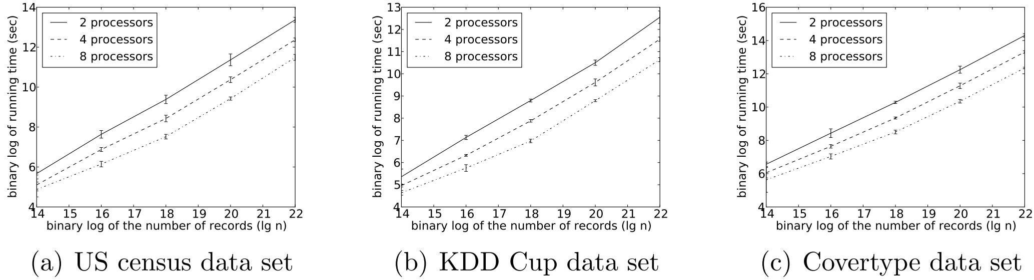
Programs were implemented in Python. As described in Section 4, $\texttt{pmda}_k$ computes $\texttt{count}(p)$ by sending the same target query (i.e., a list of attribute-value pairs) to k databases each of which runs in one of k different processes independently. On receiving all counts, $\texttt{pmda}_k$ computes the sum and uses this in further computations. Recorded running time is overall running time and includes not only database query time but also communication time and other operating system overhead.

Results

We ran the parallel algorithms $\texttt{pmda}_k, k = 2, 4, 8$ on the three data sets with different sizes (e.g., number of records). The records in each data set were randomly sampled from the original, and they were inserted into Sqlite3 databases. For each k, we repeated the algorithm for 10 times. The average running time are showed in Figure 1(a), 1(b) and 1(c).

We also compared the speedups obtained by using different numbers of processors in Table 2. Let $S_{i,j} = T_i/T_j$ be the speedup of using j processors versus using i processors where T_l represents the running time of the parallel algorithm using l processors. The results are summarized in Figure 1 and Table 2.

For a theoretical speedup s, Amdahl's law gives the maximum expected



(a) US census data set (b) KDD Cup data set (c) Covertype data set

Figure 1: Average running times (in binary log) of the parallel algorithm $\mathtt{pmda}_k$ run in $k(k = 2, 4, 8)$ processes. The error bars show 95% confidence intervals.

Data size	US Census		KDD Cup 99		Covertype	
	$S_{2,4}$	$S_{4,8}$	$S_{2,4}$	$S_{4,8}$	$S_{2,4}$	$S_{4,8}$
2^{16}	1.70	1.66	1.75	1.48	1.73	1.52
2^{18}	1.94	1.88	1.89	1.87	1.92	1.79
2^{20}	1.99	1.93	1.85	1.77	1.96	1.90
2^{22}	1.99	1.86	1.99	1.90	1.99	1.94

Table 2: Speedups of the parallel algorithms using different number of processors running on the three experimental data sets with different sizes

improvement to an overall system when only fraction p of the system is improved [1]. Using Amdahl's law, we can formulate the expectation of $S_{s,2s}$ as

$$E_p[S_{s,2s}] = \frac{(2\,p - 2)\ s - 2\,p}{(2\,p - 2)\ s - p} \tag{6}$$

For $p = 0.98$ we get $E_{0.98}[S_{2,4}] = 1.92$, and $E_{0.98}[S_{4,8}] = 1.86$. These numbers approximate our observed speedups for size 2^{22} databases. In general, solving (6) for p and using our observed values, we obtain values for p given in Table 3, which suggest that the portion of computations that are performed in parallel increase with data size. Furthermore, these results suggest that we are achieving

Size	2^{16}	2^{18}	2^{20}	2^{22}
p	0.910	0.978	0.981	0.992

Table 3: Estimated fraction of computations done in parallel at optimal speedup as a function of data size.

improvement close to the theoretical maximum for large portions of the performed computations.

6 Summary and Discussion

Count queries are well suited for distribution. We show that using a simple tree type topology network allows us to compute queries on n data points partitioned among k stores in $O(n/k + \log k)$ time.

We further show that for sequential algorithms that can be reformulated in terms of count queries, including the class of algorithms that operate on histograms, significant speedups can be achieved by the use of distributed count queries. We did this by reformulating an algorithm originally devised for haplotype tagging by single nucleotide polymorphisms in terms of count queries using a tree topology network. Used like this, distributed count queries can offer parallelization of sequential algorithms that are not easy to optimize otherwise.

The experimental results suggest that we achieve close to optimal speedups for large fractions (> 0.9) of the performed computational work. This fraction increases as data size increases, indicating increased benefit of parallelization as data size increases.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.

[2] G. E. Blelloch, R. Peng, and K. Tangwongsan. Linear-work greedy parallel approximate set cover and variants. In R. Rajaraman and F. M. auf der Heide, editors, *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 23–32. ACM, 2011.

[3] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, Dec. 2011.

[4] C. J. Date. *An Introduction to Database Systems, Eighth Edition.* Addison Wesley, July 2003.

[5] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Conference on Theory of Cryptography*, 2006.

[6] V. Grebinski and G. Kucherov. Optimal reconstruction of graphs under the additive model. In *ESA '97: Proceedings of the 5th Annual European Symposium on Algorithms*, pages 246–258, London, UK, 1997. Springer-Verlag.

[7] R. K. Guy and R. J. Nowakowski. Coin-Weighing problems. *The American Mathematical Monthly*, 102(2):164–167, Feb. 1995.

[8] R. Kaushik, J. F. Naughton, R. Ramakrishnan, and V. T. Chakravarthy. Synopses for Query Optimization: A Space-complexity Perspective. *ACM Trans. Database Syst.*, 30(4):1102–1127, Dec. 2005.

[9] M. Kearns. Efficient noise-tolerant learning from statistical queries. *Journal of ACM*, 1998.

[10] Z. Pawlak. Rough sets. *International Journal of Information and Computer Science*, 11, 1982.

[11] J. Que and F.-C. Tsui. Rank-based spatial clustering: an algorithm for rapid outbreak detection. *J Am Med Inform Assoc*, 18:123–143, 2011.

[12] B. G. Richter and D. P. Sexton. Managing and analyzing Next-Generation sequence data. *PLoS Comput Biol*, 5(6):e1000369, June 2009.

[13] F. C. Tsui, J. U. Espino, Y. Weng, A. Choudary, H. D. Su, and M. M. Wagner. Key design elements of a data utility for national biosurveillance: event-driven architecture, caching, and web service model. In *AMIA Annual Symposium Proceedings*, pages 739–743, 2005.

[14] S. Vinterbo and A. Øhrn. Minimal approximate hitting sets and rule templates. *International Journal of Approximate Reasoning*, 25(2):123–143, 2000.

[15] S. A. Vinterbo, S. Dreiseitl, and L. Ohno-Machado. Approximation properties of haplotype tagging. *BMC Bioinformatics*, 7:8, 2006.

[16] S. A. Vinterbo, E.-Y. Kim, and L. Ohno-Machado. Small, fuzzy and interpretable gene expression based classifiers. *Bioinformatics*, 21(9):1964–1970, Jan 2005.