Finding Interesting Associations without Support Pruning

Edith Cohen Mayur Datar Shinji Fujiwara Aristides Gionis Piotr Indyk
Rajeev Motwani Jeffrey D. Ullman Cheng Yang

July 17, 2000

Abstract

Association-rule mining has heretofore relied on the condition of high support to do its work efficiently. In particular, the well-known a-priori algorithm is only effective when the only rules of interest are relationships that occur very frequently. However, there are a number of applications, such as data mining, identification of similar web documents, clustering, and collaborative filtering, where the rules of interest have comparatively few instances in the data. In these cases, we must look for highly correlated items, or possibly even causal relationships between infrequent items. We develop a family of algorithms for solving this problem, employing a combination of random sampling and hashing techniques. We provide analysis of the algorithms developed, and conduct experiments on real and synthetic data to obtain a comparative performance analysis.

Keywords: Data Mining, Association Rules, Similarity Metric, Min Hashing, Locality Sensitive Hashing.

1 Introduction

A prevalent problem in large-scale data mining is that of *association-rule mining*, first introduced by Agrawal, Imielinski, and Swami [1]. This challenge is sometimes referred to as the *market-basket* problem due to its origins in the study of consumer purchasing patterns in retail stores, but the applications extend far beyond this specific setting. Suppose we have a relation R containing n tuples over a set of *boolean* attributes $A_1, A_2, \ldots, A_m$. Let $I = \{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\}$ and $J = \{A_{j_1}, A_{j_2}, \ldots, A_{j_l}\}$ be two sets of attributes. We say that $I \Rightarrow J$ is an *association rule* if the following two conditions are satisfied: *support* — the set $I \cup J$ appears in at least an s-fraction of the tuples; and, *confidence* — amongst the tuples in which I appears, at least a c-fraction also have J appearing in them. The goal is to identify all valid association rules for a given relation.

To some extent, the relative popularity of this problem can be attributed to its paradigmatic nature, the simplicity of the problem statement, and its wide applicability in identifying hidden patterns in data from more general applications than the original market-basket motivation. Arguably though, this success has as much to do with the availability of a surprisingly efficient algorithm, the lack of which has stymied other models of pattern-discovery in data mining. The algorithmic efficiency derives from an idea due to Agrawal et al [1, 2], called *a-priori*, which exploits the support requirement for association rules. The key observation is that if a set of attributes S appears in a fraction s of the tuples, then any subset of S also appears in a fraction s of the tuples.

This principle enables the following approach based on pruning: to determine a list L_k of all k-sets of attributes with high support, first compute a list L_{k-1} of all $(k-1)$-sets of attributes of high support, and consider as *candidates* for L_k only those k-sets that have all their $(k-1)$-subsets in L_{k-1}. Variants and enhancements of this approach underlie essentially all known efficient algorithms for computing association rules or their variants. Note that in the worst case, the problem of computing association rules requires time exponential in m, but the a-priori algorithm avoids this pathology on real data sets. Observe also that the confidence requirement plays no role in the algorithm, and indeed is completely ignored until the end-game, when the high-support sets are screened for high confidence.

Our work is motivated by the long-standing open question of devising an efficient algorithm for finding rules that have very high confidence, *but for which there is no (or extremely weak) support.* For example, in market-basket data the standard association-rule algorithms may be useful for commonly-purchased (i.e., high-support) items such as "beer and diapers," but are essentially useless for discovering rules such as that "Beluga caviar and Ketel vodka" are almost always bought together, because there are few people who purchase either of the two items. We develop a body of techniques which rely on the confidence requirement alone to obtain efficient algorithms.

One motivation for seeking such associations, with high-confidence but without any support requirement, is that most rules with high-support are obvious and well-known, and it is the rules of low-support that provide interesting *new* insights. Not only are the support-free associations a natural class of patterns for data mining in their own right, they also arise in a variety of applications such as: *copy detection* — identifying identical or similar documents and web pages [4, 13]; *clustering* — identifying similar vectors in high-dimensional spaces for the purposes of clustering data [6, 9]; and, *collaborative filtering* — tracking user behavior and making recommendations to individuals based on similarity of their preferences to those of other users [8, 16]. Note that each of these applications can be formulated in terms of a table whose columns tend to be sparse, and the goal is to identify column pairs that appear to be similar, without any support requirement.

The notion of confidence is asymmetric or uni-directional, and it will be convenient for our purpose to work with a symmetric or bi-directional measure of interest. At a conceptual level, we view the data as a 0/1 matrix M with n rows and m columns. Typically, the matrix is fairly sparse and we assume that the *average* number of 1s per row is r and that $r << m$. (For the applications we have in mind, n could be as much as 10^9, m could be as large as 10^6, and r could be as small as 10^2). Define C_i as the set of rows that have a 1 in column c_i; also, define the *density* of column c_i as $d_i = |C_i|/n$. We define the *similarity* of two columns c_i and c_j as

$$S(c_i, c_j) = \frac{|C_i \cap C_j|}{|C_i \cup C_j|}.$$

That is, the similarity of c_i and c_j is the fraction of rows, amongst those containing a 1 in either c_i or c_j, that contain a 1 in both c_i and c_j. Observe that the definition of similarity is symmetric with respect to c_i and c_j; in contrast, the confidence of the rule $\{c_i\} \Rightarrow \{c_j\}$ is given by

$$\mathrm{Conf}(c_i, c_j) = \frac{|C_i \cap C_j|}{|C_i|}.$$

To identify all pairs of columns with similarity exceeding a respecified threshold is easy when the matrix M is small and fits in main memory, since a brute-force enumeration algorithm requires $O(m^2 n)$ time. We are more interested in the case where M is large and the data is disk-resident.

In this paper, our primary focus is on the problem of identifying all *pairs of columns* with similarity exceeding a pre-specified threshold s^*. Restricting ourselves to this most basic version of the problem will enable us clearly to showcase our techniques for dealing with the main issue of achieving algorithmic efficiency in the absence of the support requirement. It is possible to generalize our techniques to more complex settings, and we discuss this briefly before moving on to the techniques themselves. It will be easy to verify that our basic approach generalizes to the problem of identifying high-confidence association rules on pairs of columns, as discussed in Section 7. It should be noted that several recent papers [3, 14, 15] have expressed dissatisfaction with the use of confidence as a measure of interest for association rules and have suggested various alternate measures. Our ideas are applicable to these new measures of interest as well. A major restriction in our work is that we only deal with pairs of columns. However, we believe that it should be possible to apply our techniques to the identification of more complex rules; this matter is discussed in more detail in Section 7.

All our algorithms for identifying pairs of similar columns follow a very natural three-phase approach: *compute signatures*, *generate candidates*, and *prune candidates*. In the first phase, we make a pass over the table generating a small hash-signature for each column. Our goal is to deal with large-scale tables sitting in secondary memory, and this phase produces a "summary" of the table that will fit into main memory. In the second phase, we operate in main memory, generating candidate pairs from the column signatures. Finally, in the third phase, we make another pass over the original table, determining for each candidate pair whether it indeed has high similarity. The last phase is identical in all our algorithms: while scanning the table data, maintain for each candidate column-pair (c_i, c_j) the counts of the number of rows having a 1 in at least one of the two columns and also the number of rows having a 1 in both columns. Consequently, we limit the ensuing discussion to the proper implementation of only the first two phases.

The key ingredient, of course, is the hashing scheme for computing signatures. On the one hand, it needs to be extremely fast, produce small signatures, and be able to do so in a single pass over the data. Competing with this goal is the requirement that there are not too many *false-positives*, i.e., candidate pairs that are not really highly-similar, since the time required for the third phase depends on the number of candidates to be screened. A related requirement is that there are extremely few (ideally, none) *false-negatives*, i.e., highly-similar pairs that do not make it to the list of candidates.

In Section 3 we present a family of schemes based on a technique called *Min-Hashing (MH)* which is inspired by an idea used by Cohen [5] to estimate the size of transitive closure and reachability sets (see also Broder [4]). The idea is to implicitly define a random order on the rows, selecting for each column a signature that consists of the first row index (under the ordering) in which the column has a 1. We will show that the probability that two columns have the same signature is proportional to their similarity. To reduce the probability of false-positives and false-negatives, we can collect k signatures by independently repeating the basic process or by picking the first k rows in which the column has 1's. The main feature of the Min-Hashing scheme is that, for a suitably large choice of k, the number of false-positives is fairly small and the number of false-negatives is essentially zero. A disadvantage is that as k rises, the space and time required for the second phase (candidate generation) increases.

Our second family of schemes, called *Locality-Sensitive Hashing (LSH)*, is presented in Section 4 and is inspired by the ideas used by Gionis, Indyk, and Motwani [7] for high-dimensional nearest neighbors (see also Indyk and Motwani [11]). The basic idea here is to implicitly partition the set of rows, computing a signature based on the pattern of 1's of a column in each subtable; for example, we could just compute a bit for each column in a subtable, denoting whether the number of 1's in the column is greater than zero or not. This family of schemes suffers from the disadvantage that reducing the number of false-positives increases the number of false-negatives, and vice versa, unlike in the previous scheme. While it tends to produce more false-positives or false-negatives, it has the advantage of having much lower space and time requirements than Min-Hashing.

We have conducted extensive experiments on both real and synthetic data, and the results are presented in Section 5. As expected, the experiments indicate that our schemes outperform the a-priori algorithm by orders of magnitude. They also illustrate the point made above about the trade-off between accuracy and speed in our two algorithms. If it is important to avoid any false-negatives, than we recommend the use of the Min-Hashing schemes which tend to be slower. However, if speed is more important than complete accuracy in generating rules, than the Locality-Sensitive Hashing schemes are to be preferred. In Section 7 we discuss the extensions of our work, and we provide some interesting directions for future work, and finally, we conclude in Section 8 by summarizing our algorithms and their distinctive features.

2 Motivation

Our main contribution has been to develop techniques to mine low support rules which cannot be done effectively with the existing techniques namely a-priori.

There are a number of applications like clustering, copy detection, and collaborative filtering where the low support rules are of significance.

We have looked into one such application and we report here our findings. We applied our algorithms to mine for pairs of words that occur together in news documents to check if they provide any interesting information. The news articles were obtained from Reuters Press. Indeed the similar pairs provide some very interesting information as can be seen from a few examples that we provide in figure 1. Most of the pairs are names of famous international personalities, cities, terms from medicine and other fields, phrases from foreign languages and other miscellaneous items like author-book pairs, organizations etc. We also get clusters of words, i.e. groups of words for which most of the pairs in the group have high similarity. Such an example is the cluster (chess, Timman, Karpov, Soviet, Ivanchuk, Polger) that stands for a chess event. It should be noted that these pairs have very low support. A running time comparison between a-priori and our algorithms is provided in section 5.

Names	Terminology	Phrases	Misc Relations
(Dalai, Lama)	(pneumocystis, carinii)	(avant, garde)	(encyclopedia, Britannica)
(Meryl, Streep)	(meseo, oceania)	(mache, papier)	(Salman, Satanic)
(Bertolt, Brecht)	(fibrosis, cystic)	(cosa, nostra)	(Mardi, Gras)
(Buenos, Aires)		(hors, oeuvres)	(emperor, Hirohito)
(Darth, Vader)		(presse, agence)	

Figure 1: Examples of different types of similar pairs found in the news articles

3 Min-Hashing Schemes

The Min-Hashing scheme used an idea due to Cohen [5], in the context of estimating transitive closure and reachability sets. The basic idea in the Min-Hashing scheme is to randomly permute the rows and for each column c_i compute its hash value $h(c_i)$ as the index of the first row under the permutation that has a 1 in that column. For reasons of efficiency, we do not wish to explicitly permute the rows, and indeed would like to compute the hash value for each column in a single pass over the table. To this end, while scanning the rows, we will simply associate with each row a hash value that is a number chosen independently and uniformly at random from a range R. Assuming that the number of rows is no more than 2^{16}, it will suffice to choose the hash value as a random 32-bit integer, avoiding the "birthday paradox" [12] of having two rows get identical hash value. Furthermore, while scanning the table and assigning random hash values to the rows, for each column c_i we keep track of the *minimum* hash value of the rows which contain a 1 in that column. Thus, we obtain the Min-Hash value $h(c_i)$ for each column c_i in a single pass over the table, using $O(m)$ memory.

Proposition 1 *For any column pair* (c_i, c_j), $\text{Prob}[h(c_i) = h(c_j)] = \dfrac{|C_i \cap C_j|}{|C_i \cup C_j|} = S(c_i, c_j)$

This is easy to see since two columns will have the same Min-Hash value if and only if, in the random permutation of rows defined by their hash values, the first row with a 1 in column c_i is also the first row with a 1 in column c_j. In other words, $h(c_i) = h(c_j)$ if and only if in restriction of the permutation to the rows in $C_i \cup C_j$, the first row belongs to $C_i \cap C_j$. Since each row has the same probability of coming first in the random permutation, the probability of the event $h(c_i) = h(c_j)$ is exactly the cardinality of the set $C_i \cap C_j$ over the cardinality of the set $C_i \cup C_j$, and this is exactly

$S(c_i, c_j)$.

In order to be able to determine the degree of similarity between column-pairs, it will be necessary to determine multiple (say k) independent Min-Hash values for each column. To this end, in a single pass over the input table we select (in parallel) k independent hash values for each row, defining k distinct permutations over the rows. Using $O(mk)$ memory, during the single pass we can also determine the corresponding k Min-Hash values, say $h_1(c_j), \ldots, h_k(c_j)$, for each column c_j under each of k row permutations. In effect, we obtain a matrix $\widehat{M}$ with k rows, m columns, and $\widehat{M}_{ij} = h_i(c_j)$, where the k entries in a column are the Min-Hash values for it. The matrix $\widehat{M}$ can be viewed as a compact representation of the matrix M. We will show in Theorem 1 below that the similarity of column-pairs in M is captured by their similarity in $\widehat{M}$.

Definition 1 *Let $\widehat{S}(c_i, c_j)$ be the fraction of Min-Hash values that are identical for c_i and c_j, i.e.,*

$$\widehat{S}(c_i, c_j) = \frac{|\{l \mid 1 \leq l \leq k \text{ and } \widehat{M}_{li} = \widehat{M}_{lj}\}|}{k} = \frac{|\{l \mid 1 \leq l \leq k \text{ and } h_l(c_i) = h_l(c_j)\}|}{k}.$$

Example 1 *To give a simple illustration of the above ideas, consider the following boolean matrix*

$$M = \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array}$$

and two Min-Hash functions h_1 and h_2, defined through the permutations

$$\pi_1 = \{1 \to 3, 2 \to 1, 3 \to 2, 4 \to 4\} \text{ and}$$

$$\pi_2 = \{1 \to 2, 2 \to 4, 3 \to 3, 4 \to 1\}$$

The notation $i \to j$ denotes that the permutation π maps the i-th row to the j-th. We concentrate for a moment on the column c_1: The first row under the permutation π_1 that has a 1 is the row r_2, so $h_1(c_1) = 2$. Similarly, the row r_1 is the first row with a 1 under the permutation π_2, and therefore $h_2(c_1) = 1$. Working similarly for the other two columns we get

$$\widehat{M} = \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 2 & 2 & 3 \\ 1 & 1 & 4 \end{bmatrix} \end{array} \begin{array}{c} \\ h_1 \\ h_2 \end{array}$$

The matrix $\widehat{M}$ has smaller size than the original matrix M, but still the pairwise similarities between the columns are roughly maintained. In particular, we have $S(c_1, c_2) = 2/3$, $S(c_1, c_3) = 0$, and $S(c_2, c_3) = 1/4$, while $\widehat{S}(c_1, c_2) = 1$, $\widehat{S}(c_1, c_3) = 0$, and $\widehat{S}(c_2, c_3) = 0$, respectively. In this toy-example, because $\widehat{M}$ has only 2 rows, the approximation to the actual similarities is very coarse, but as we will see later by considering more hash functions we can boost the accuracy of the estimation arbitrarily high. ∎

We have defined $\widehat{S}(c_i, c_j)$ as the fraction of rows of $\widehat{S}$ in which the Min-Hash entries for columns c_i and c_j are identical. We now show that $\widehat{S}(c_i, c_j)$ is a good estimator of $S(c_i, c_j)$. Recall that we set a threshold s^* such that two columns are said to be *highly-similar* if $S(c_i, c_j) \geq s^*$. Assume that s^* is lower bounded by some constant c. The following theorem shows that we are unlikely to get too many false-positives and false-negatives by using $\widehat{S}$ to determine similarity of column-pairs in the original matrix M.

Theorem 1 *Let $0 < \delta < 1$, $\epsilon > 0$, and $k \geq 2\delta^{-2}c^{-1}\log\epsilon^{-1}$. Then, for all pairs of columns c_i and c_j, we have the following two properties.*

a) *If $S(c_j, c_j) \geq s^* \geq c$, then $\widehat{S}(c_i, c_j) \geq (1-\delta)s^*$ with probability at least $1-\epsilon$.*

b) *If $S(c_j, c_j) \leq c$, then $\widehat{S}(c_i, c_j) \leq (1+\delta)c$ with probability at least $1-\epsilon$.*

We sketch the proof of the first part of the theorem; the proof of the second part is quite similar and is omitted. Fix any two columns c_i and c_j having similarity $S(c_j, c_j) \geq s^*$. Let X_l be a random variable that takes on value 1 if $h_l(c_i) = h_l(c_j)$, and value 0 otherwise; define $X = X_1 + \ldots + X_k$. By Proposition 1, $E[X_l] = S(c_i, c_j) \geq s^*$; therefore, $E[X] \geq ks^*$. Applying the Chernoff bound [12] with the random variable X, we obtain that

$$\text{Prob}[X < (1-\delta)ks^*] \leq \text{Prob}[X < (1-\delta)E[X]] \leq e^{-\frac{\delta^2 E[X]}{2}} \leq e^{-\frac{\delta^2 ks^*}{2}} \leq e^{-\frac{\delta^2 kc}{2}} < \epsilon.$$

To establish the first part of the theorem, simply notice that $\widehat{S}(c_i, c_j) = X/k$.

Theorem 1 establishes that for sufficiently large k, if two columns have high similarity (at least s^*) in M then they agree on a correspondingly large fraction of the Min-Hash values in $\widehat{M}$; conversely, if their similarity is low (at most c) in M then they agree on a correspondingly small fraction of the Min-Hash values in $\widehat{M}$. Since $\widehat{M}$ can be computed in a single pass over the data using $O(km)$ space, we obtain the desired implementation of the first phase (signature computation). We now turn to the task of devising a suitable implementation of the second phase (candidate generation).

3.1 Candidate Generation from Min-Hash Values

Having computed the signatures in the first phase as discussed in the previous section, we now wish to generate the candidate column-pairs in the second phase. At this point, we have a $k \times m$ matrix $\widehat{M}$ containing k Min-Hash values for each column. Since $k << n$, we assume the $\widehat{M}$ is much smaller than the original data and fits in main memory. The goal is to identify all column-pairs which agree in a large enough fraction (at least to $(1-\delta)s^*$) of their Min-Hash values in $\widehat{M}$. A brute-force enumeration will require $O(k)$ time for each column-pair, for a total of $O(km^2)$. We present two techniques that avoid the quadratic dependence on m and are considerably faster when (as is typically the case) the *average similarity* $\overline{S} = \sum_{1 \leq i,j \leq m} S(c_i, c_j)/m^2$ is low.

Row-Sorting: For this algorithm, view the rows of $\widehat{M}$ as a list of tuples containing a Min-Hash value and the corresponding column number. We sort each row on the basis of the Min-Hash values. This groups together identical Min-Hash values into a sequence of "runs." We maintain for each column an index into the position of its Min-Hash value in each sorted row. To estimate the similarity of column c_i with all other columns, we use the following algorithm: use m counters for column c_i where the jth counter stores the number of rows in which the Min-Hash values of columns c_i and c_j are identical; for each row $1, \ldots, k$, index into the run containing the Min-Hash value for c_i, and for each other column represented in this run, increment the corresponding counter. To avoid $O(m^2)$ counter initializations, we re-use the same $O(m)$ counters when processing different columns, and remember and re-initialize only counters that were incremented at least once. We estimate the running time of this algorithm as follows. Sorting the rows requires total time $O(km \log m)$; thereafter, indexes on the columns can be built in time $O(km)$. The remaining time amounts to the total number of counter increments. When processing a row with column c_i, the number of counter increments is in fact the length of a run. The *expected* length of a run equals the sum of similarities $\sum_{j=1}^{m} S(c_i, c_j)$. Hence, the expected counter-increment cost when processing c_i is $O(k \sum_{j=1}^{m} S(c_i, c_j))$, and the expected combined increments cost is $O(k \sum_{1 \leq i,j \leq m} S(c_i, c_j)) = O(k\overline{S}m^2)$. Thus, the expected total time required for this algorithm is is $O(km \log m + km^2\overline{S})$. Note that the average similarity $\overline{S}$ is typically a small fraction, and so the latter term in the running time is not really quadratic in m as it appears to be.

Hash-Count: The next section introduces the K-Min-Hashing algorithm where the signatures for each column c_i is a set $\textsc{Sig}_i$ of at most, but not exactly, k Min-Hash values. The similarity of a column-pair (c_i, c_j) is then estimated by computing the size of $\textsc{Sig}_i \cap \textsc{Sig}_j$; clearly, it suffices to consider ordered pairs (c_j, c_i) such that $j < i$. This task can be accomplished via the following hash-count algorithm. We associate a bucket with each Min-Hash value. Buckets are indexed using a hash function defined over the Min-Hash values, and store column-indexes for all columns c_i with some element of $\textsc{Sig}_i$ hashing into that bucket. We consider the columns $c_1, c_2, \ldots, c_m$ in order, and for column c_i we use $i-1$ counters, of which of the jth counter stores $\textsc{Sig}_j \cap \textsc{Sig}_i$. For each Min-Hash value $v \in \textsc{Sig}_i$, we access its hash-bucket and find the indexes of all columns c_j $(j < i)$ which have $v \in \textsc{Sig}_j$. For each column c_j in the bucket, we increment the counter for (c_j, c_i). Finally, we add c_i itself to the bucket.

Hash-Count can be easily adapted for use with the original Min-Hash scheme where we instead want to compute for each pair of columns the number of $\widehat{M}$ rows in which the two columns agree. To this end, we use a different hash table (and set of buckets) for each row of the matrix $\widehat{M}$, and execute the same process as for K-Min-Hash. The argument used for the row-sorting algorithm shows that hash-count for Min-Hashing takes $O(k\overline{S}m^2)$ time. The running time of Hash-Count for K-Min-Hash amounts to the number of counter increments. The number of increments made to a counter $(c_i, c_j))$ is exactly the size of $|\textsc{Sig}_i \cap \textsc{Sig}_j|$. A simple argument (see Lemma 1) shows that the expected size $E\{|\textsc{Sig}_i \cap \textsc{Sig}_j|\}$ is between $\min\{k, |C_i \cup C_j|\} S(c_i, c_j)$ and $\min\{2k, |C_i \cup C_j|\} S(c_i, c_j)$. Thus, the expected total running time of the hash-table scheme is $O(k\overline{S}m^2)$ in both cases.

3.2 The K-Min-Hashing Algorithm

One disadvantage of the Min-Hashing scheme outlined above is that choosing k independent Min-Hash values for each column entailed computing k independent hash values for each row of the matrix M. This has a negative effect on the efficiency of the signature-computation phase. On the other hand, using k Min-Hash values per column is essential for reducing the number of false-positives and false-negatives. We now present a modification called K-Min-Hashing (K-MH) in which we use only a single hash value for each row, setting the k Min-Hash values for each column to be the hash values of the first k rows (under the induced row permutation) containing a 1 in that column. (A similar approach was also mentioned in [5] but without an analysis.) In other words, for each column we pick the k smallest hash values for the rows containing a one in that column. If a column c_i has fewer 1s than k, we assign as Min-Hash values all hash values corresponding to rows with 1s in that column. The resulting set of (at most) k Min-Hash values forms the signature of the column c_i and is denoted by $\textsc{Sig}_i$.

Proposition 2 *In the K-Min-Hashing scheme, for any column c_i, the signature $\textsc{Sig}_i$ consists of the hash values for a uniform random sample of distinct rows from C_i.*

We remark that if the number of 1s in each column is significantly larger than k, then the hash values may be considered independent and the analysis from Min-Hashing applies. The situation is slightly more complex when the columns are sparse, which is the case of interest to us.

Let $\textsc{Sig}_{i\cup j}$ denote the k smallest elements of $C_i \cup C_j$; if $|C_i \cup C_j| < k$ then $\textsc{Sig}_{i\cup j} = C_i \cup C_j$. We can view $\textsc{Sig}_{i\cup j}$ as the signature of the "column" that would correspond to $C_i \cup C_j$. Observe that $\textsc{Sig}_{i\cup j}$ can be obtained (in $O(k)$ time) from $\textsc{Sig}_i$ and $\textsc{Sig}_j$ since it is in fact the set of the smallest k elements from $\textsc{Sig}_i \cup \textsc{Sig}_j$. Since $\textsc{Sig}_{i\cup j}$ corresponds to a set of rows selected uniformly at random from all elements of $C_i \cup C_j$, the expected number of elements of $\textsc{Sig}_{i\cup j}$ that belong to the subset $C_i \cap C_j$ is exactly $|\textsc{Sig}_{i\cup j}| \times |C_i \cap C_j| / |C_i \cup C_j| = |\textsc{Sig}_{i\cup j}| \times S(c_i, c_j)$. Also, $\textsc{Sig}_{i\cup j} \cap C_i \cap C_j = \textsc{Sig}_{i\cup j} \cap \textsc{Sig}_i \cap \textsc{Sig}_j$, since the signatures are just the smallest k elements. Hence, we obtain the following theorem.

Theorem 2 *An unbiased estimator of the similarity $S(c_i, c_j)$ is given by the expression*

$$\frac{|\textsc{Sig}_{i\cup j} \cap \textsc{Sig}_i \cap \textsc{Sig}_j|}{|\textsc{Sig}_{i\cup j}|}.$$

Consider the computational cost of this algorithm. While scanning the data, we generate one hash value per row, and for each column we maintain the minimum k hash values from those corresponding to rows that contain 1 in that column. We maintain the k minimum hash values for each column in a simple data structure that allows us to insert a new value (smaller than the current maximum) and delete the current maximum in $O(\log k)$ time. The data structure also makes the maximum element amongst the k current Min-Hash values of each column readily available. Hence, the computation for each row is constant time for each 1 entry and additional $\log k$ time for each column with 1 entry where the hash value of the row was amongst the k smallest seen so far. A simple probabilistic argument shows that the expected number of rows on which the k-Min-Hash list of a column c_i gets updated is $O(k \log |C_i|) = O(k \log n)$. It follows that the total computation cost is a single scan of the data and $O(|M| + mk \log n \log k)$, where $|M|$ is the number of 1s in the matrix M.

In the second phase, while generating candidates, we need to compute the sets $\textsc{Sig}_{i\cup j}$ for each column-pair using merge join ($O(k)$ operations) and while we are merging we can also find the elements that belong to $\textsc{Sig}_i \cap \textsc{Sig}_j$. Hence, the total time for this phase is $O(km^2)$. The quadratic dependence on the number of columns is prohibitive and is caused by the need to compute $\textsc{Sig}_{i\cup j}$ for each column-pair. Instead, we first apply a considerably more efficient *biased approximate* estimator for the similarity. The biased estimator is computed for all pairs of columns using Hash-Count in $O(k\overline{S}m^2)$ time. Next we perform a *main-memory candidate pruning* phase, where the unbiased estimator of Theorem 2 is explicitly computed for all pairs of columns where the approximate biased estimator exceeds a threshold.

The choice of threshold for the biased estimator is guided by the following lemma.

Lemma 1

$$E\{|\textsc{Sig}_i \cap \textsc{Sig}_j|\}/\min\{2k, |C_i \cup C_j|\} \leq S(c_i, c_j) \leq E\{|\textsc{Sig}_i \cap \textsc{Sig}_j|\}/\min\{k, |C_i \cup C_j|\} \ .$$

Alternatively, the biased estimator and choice of threshold can be derived from the following analysis. Let (c_i, c_j) be a column-pair with $|C_i| \geq |C_j|$; define $C_{ij} = C_i \cap C_j$. As before, for each column c_i, we choose a set $\textsc{Sig}_i$ of k Min-Hash values. Let $\textsc{Sig}_{ij} = \textsc{Sig}_i \cap C_{ij}$ and $\textsc{Sig}_{ji} = \textsc{Sig}_j \cap C_{ij}$. Then, the expected sizes of $\textsc{Sig}_{ij}$ and $\textsc{Sig}_{ji}$ are given by $k|C_{ij}|/|C_i|$ and $k|C_{ij}|/|C_j|$. Also, $|\textsc{Sig}_i \cap \textsc{Sig}_j| = min(|\textsc{Sig}_{ij}|, |\textsc{Sig}_{ji}|)$. Hence can compute the expected value as

$$E[|\textsc{Sig}_i \cap \textsc{Sig}_j|] = \sum_{x=0}^{k}\sum_{y=0}^{k} \text{Prob}[|\textsc{Sig}_{ij}| = x]\text{Prob}[|\textsc{Sig}_{ji}| = y \mid |\textsc{Sig}_{ij}| = x] \min(x, y).$$

Since $|C_i| > |C_j|$, we have $E(|\textsc{Sig}_{ij}|) \leq E(|\textsc{Sig}_{ji}|)$. We assume that $\text{Prob}[|\textsc{Sig}_{ij}| > |\textsc{Sig}_{ji}|] \approx 0$ or $\sum_{y=x}^{k} P[|\textsc{Sig}_{ji}| = y \mid |\textsc{Sig}_{ij}| = x] \approx 1$. Then, the above equation becomes

$$\begin{aligned}
E[|\textsc{Sig}_i \cap \textsc{Sig}_j|] &= \sum_{x=0}^{k}\sum_{y=x}^{k} \text{Prob}[|\textsc{Sig}_{ij}| = x]\text{Prob}[|\textsc{Sig}_{ji}| = y \mid |\textsc{Sig}_{ij}| = x]x \\
&= \sum_{x=0}^{k} \text{Prob}[|\textsc{Sig}_{ij}| = x]x \sum_{y=x}^{k} \text{Prob}[|\textsc{Sig}_{ji}| = y \mid |\textsc{Sig}_{ij}| = x] \\
&= E[\textsc{Sig}_{ij}].
\end{aligned}$$

Thus, we obtain the estimator $E[|\textsc{Sig}_i \cap \textsc{Sig}_j|] \approx k|C_{ij}|/|C_i|$. We use this estimate to calculate $|C_{ij}|$ and use that to estimate the similarity since we know $|C_i|$ and $|C_j|$. We compute $|\textsc{Sig}_i \cap \textsc{Sig}_j|$ using the hash table technique that we have described earlier in Section 3.1. The time required to compute the hash values is $O(|M| + mk \log n \log k)$ as described earlier, and the time for computing $|\textsc{Sig}_i \cap \textsc{Sig}_j|$ is $O(k\overline{S}m^2)$.

4 Locality-Sensitive Hashing Schemes

In this section we show how to obtain a significant improvement in the running time with respect to the previous algorithms by resorting to *Locality Sensitive Hashing (LSH)* technique introduced by Indyk and Motwani [11] in designing main-memory algorithms for nearest neighbor search in high-dimensional Euclidean spaces; it has been subsequently improved and tested in [7]. We apply the LSH framework to the Min-Hash functions described in earlier section, obtaining an algorithm for similar column-pairs. This problem differs from nearest neighbor search in that the data is known in advance. We exploit this property by showing how to optimize the running time of the algorithm given constraints on the quality of the output. Our optimization is *input-sensitive*, i.e., takes into account the characteristics of the input data set.

The key idea in LSH is to hash columns so as to ensure that for each hash function, the probability of collision is much higher for similar columns than for dissimilar ones. Subsequently, the hash table is scanned and column-pairs hashed to the same bucket are reported as similar. Since the process is probabilistic, both false positives and false negatives can occur. In order to reduce the former, LSH amplifies the difference in collision probabilities for similar and dissimilar pairs. In order to reduce false negatives, the process is repeated a few times, and the union of pairs found during all iterations are reported. The fraction of false positives and false negatives can be analytically controlled using the parameters of the algorithm.

Although not the main focus of this paper, we mention that the LSH algorithm can be adapted to the *on-line* framework of [10]. In particular, it follows from our analysis that each iteration of our algorithm reduces the number of false negatives by a fixed factor; it can also add new false positives, but they can be removed at a small additional cost. Thus, the user can monitor the progress of the algorithm and interrupt the process at any time if satisfied with the results produce so far. Moreover, the higher the similarity, the earlier the pair is likely to be discovered. Therefore, the user can terminate the process when the output produced appears to be less and less interesting.

4.1 The Min-LSH Scheme

We now present the Min-LSH (M-LSH) scheme for finding similar column-pairs from the matrix $\widehat{M}$ of Min-Hash values. The M-LSH algorithm splits the matrix $\widehat{M}$ into l sub-matrices of dimension $r \times m$. Recall that $\widehat{M}$ has dimension $k \times m$, and here we assume that $k = l \cdot r$. Then, for each of the l sub-matrices, we repeat the following. Each column, represented by the r Min-Hash values in the current sub-matrix, is hashed into a table using as hashing key the concatenation of all r values. If two columns are similar, there is a high probability that they agree in all r Min-Hash values and so they hash into the same bucket. At the end of the phase we scan the hash table and produce pairs of columns that have been hashed to the same bucket. To amplify the probability that similar columns will hash to the same bucket, we repeat the process l times. Let $P_{r,l}(c_i, c_j)$ be the probability that columns c_i and c_j will hash to the same bucket at least once; since the value of P depends only upon $s = S(c_i, c_j)$, we simplify notation by writing $P(s)$.

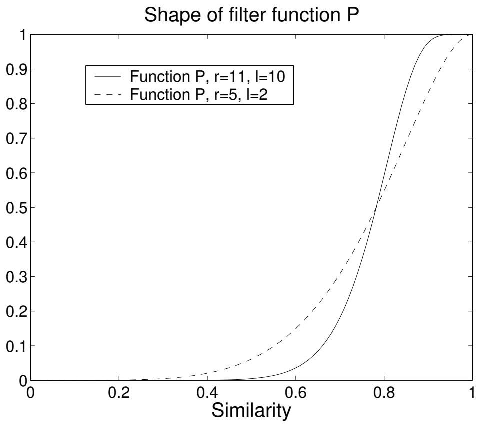
Lemma 2 *Assume that columns c_i and c_j have similarity s, and also let s^* be the similarity threshold. For any $0 < \delta, \epsilon < 1$, we can choose the parameters r and l such that:*

- *For any $s \geq (1+\delta)s^*$, $P_{r,l}(c_i, c_j) \geq 1 - \epsilon$*
- *For any $s \leq (1-\delta)s^*$, $P_{r,l}(c_i, c_j) \leq \epsilon$*

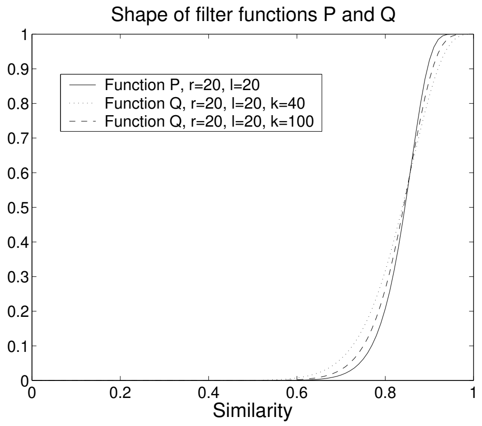
Proof: By Proposition 1, the probability that columns c_i, c_j agree on one Min-Hash value is exactly s and the probability that they agree in a group of r values is s^r. If we repeat the hashing process l times, the probability that they will hash at least once to the same bucket would be $P_{r,l}(c_i, c_j) = 1 - (1 - s^r)^l$. The lemma follows from the properties of the function P. ■

Lemma 2 states that for large values of r and l, the function $P_{r,l}$ approximates the unit step function translated to the point $C = s^*$, which can be used to filter out *all and only* the pairs with similarity at most s^*. The time/space requirements of the algorithm are proportional to $k = l \cdot r$, so the increase in the values of r and l is subject to a quality-efficiency trade-off. This is shown graphically at Figure 2(a), where for larger values of the parameters r and l we get a better approximation to the unit step function.

We also present here another scheme that *approximates* the function $P_{r,l}$ but uses k Min-Hash values where k is less than the product $l \cdot r$. The reason that we might want to do something like that is to improve the efficiency of the algorithm as the time taken to generate the signatures is linear in the number of Min-Hash values. Let us assume that the value of k is fixed and specified by the time constraints, while the product of the ideal parameters r and l exceeds k. Then, we can approximate $P_{r,l}$ by picking uniformly at random r Min-Hash values among the k available, concatenating them to create the hashing key, and repeating the process l times. Notice that some of the k Min-Hash values can participate to more than one hashing keys.



(a)



(b)

Figure 2: (a) For larger values of the parameters r and l the filter function $P_{r,l}$ provides a better approximation to the unit step function. (b) Function $Q_{r,l,k}$ is a good approximation of $P_{r,l}$ by using much less Min-Hash values. In this example, the implementation of $P_{20,20}$ would need 400 Min-Hash values, while $Q_{20,20,40}$ is a good approximation using only 40 Min-Hash values.

If s is again the similarity of two columns c_i and c_j, let $Q_{r,l,k}(s)$ be the probability that the columns c_i, c_j will hash to the same bucket at least once. If c_i and c_j agree in exactly d out of the k Min-Hash values the probability that they collide at least once is

$$q_{r,l,k}(d) = 1 - (1 - (\frac{d}{k})^r)^l$$

Summing this expression over all values of d we get

$$Q_{r,l,k}(s) = \sum_{d=1}^{k} \binom{k}{d} s^d (1-s)^{k-d} q_{r,l,k}(d)$$

The function $Q_{r,l,k}$ has similar shape with the function $P_{r,l}$. In fact, for the same values of r and l, the function $Q_{r,l,k}$ is an approximation to $P_{r,l}$, with $P_{r,l}$ always being sharper, and $Q_{r,l,k}$ becoming more and more sharp for larger values of k. An example of these filter functions is shown in Figure 2(b).

In practice, if we are willing to allow a number of false negatives (n_-) and false positives (n_+), we can determine optimal values for r and l that achieve this quality.

Specifically, assume that we are given (an estimate of) the similarity distribution of the data, defined as $distr(s_i)$ to be the number of pairs having similarity s_i. This is not an unreasonable assumption, since we can approximate this distribution by sampling a small fraction of columns and estimating all pairwise similarity. The expected number of false negatives would be $\sum_{s_i \geq s_0} distr(s_i)(1-$

$P(s_i)$), and the expected number of false positives would be $\sum_{s_i<s_0} distr(s_i)P(s_i)$. Therefore, the problem of estimating optimal parameters turns into the following minimization problem:

$$\begin{array}{lll} \text{minimize} & l \cdot r & \\ \text{subject to} & \sum_{s_i \geq s_0} dirst(s_i)(1-P(s_i)) \leq n_- \quad \text{and} & \displaystyle\sum_{s_i<s_0} distr(s_i)P(s_i) \leq n_+ \end{array}$$

This is an easy problem since we have only two parameters to optimize, and their feasible values are small integers. One approach is to solve the minimization problem by iterating on small values of r, finding a lower bound on the value of l by solving the first inequality, and then performing binary search until the second inequality is satisfied. In most experiments, the optimal value of r was between 5 and 20.

4.2 The Hamming-LSH Scheme

We now propose another scheme, Hamming-LSH (H-LSH), for finding highly-similar column-pairs. The idea is to reduce the problem to searching for column-pairs having small *Hamming distance.* Recall that the Hamming distance $d_H(x,y)$ of two vectors x and y is the number of positions on which the two vectors differ. In order to solve the latter problem we employ the techniques similar to those used in [7] to solve the nearest neighbor problem. We start by establishing the correspondence between the similarity and Hamming distance (the proof is easy).

Lemma 3 $S(C_i,C_j) = \frac{|C_i|+|C_j|-d_H(c_i,c_j)}{|C_i|+|C_j|+d_H(c_i,c_j)}$.

It follows that when we consider pairs (c_i,c_j) such that the sum $\rho = |C_i|+|C_j|$ is fixed, then the high value of $S(c_i,c_j)$ corresponds to small values of $d_H(c_i,c_j)$ and vice versa. Hence, we partition columns into groups of similar density and for each group we find pairs of columns that have small Hamming distance. First, we briefly describe how to search for pairs of columns with small Hamming distance. This scheme is similar to to the technique from [7] and can be analyzed using the tools developed in there. This scheme finds highly-similar columns assuming that the density of all columns is roughly the same. This is done by partitioning the rows of database into p subsets. For each partition, process as in the previous algorithm. We declare a pair of columns as a candidate if they agree on any subset. Thus this scheme is exactly similar to the earlier scheme, except that we are dealing with the actual data instead of Min-Hash values.

However there are two problems with this scheme. One problem is that if the matrix is sparse, most of the subsets just contain zeros and also the columns do not have similar densities as assumed. The following algorithm (which we call *H-LSH*) improves on the above basic algorithm.

The basic idea is as follows. We perform computation on a *sequence* of matrices with increasing densities; we denote them by $M_0, M_1, M_2, \ldots$. The matrix M_{i+1} is obtained from the matrix M_i by randomly pairing all rows of M_i, and placing in M_{i+1} the "OR" of each pair. [1] One can see that for each i, M_{i+1} contains half the rows of M_i (for illustration purposes we assume that the initial number of rows is a power of 2). The algorithm is applied to all matrices in the set. A pair of columns can become a candidate only on a matrix M_i in which they are both sufficiently dense and both their densities belong to a certain range. False negatives are controlled by repeating each sample l times, and taking the *union* of the candidate sets across all l runs. Hence, kr rows are extracted from each compressed matrix. Note that this operation may increase false positives.

We now present the algorithm that was implemented. Experiments show that this scheme is better than the Min-Hashing algorithms in terms of running time, but the number of false positives is much larger. Moreover, the number of false positives is increases rapidly if we try to reduce the number of false negatives. In the case of Min-Hashing algorithms, if we decreased the number of false negatives by increasing k, the number of false positives would also decrease.

The Algorithm:

1. Set $M_0 = M$ and generate $M_1, M_2, \ldots$ as described above.

2. For each $i \geq 0$, select k sets of r sample rows from M_i.

3. A column pair is a candidate if there exists an i, such that (i) the column pair has density in $(1/t, (t-1)/t)$ in M_i, and (ii) has identical hash values (essentially, identical r-bit representations) in at least one of the k runs.

Note that t is a parameter that indicates the range of density for candidate pairs, and we use $t = 4$ in our experiments.

[1]Notice, that the "OR operation" gives similar results to hashing each columns to a set of increasingly smaller hash table; this provides an alternative view of our algorithm.

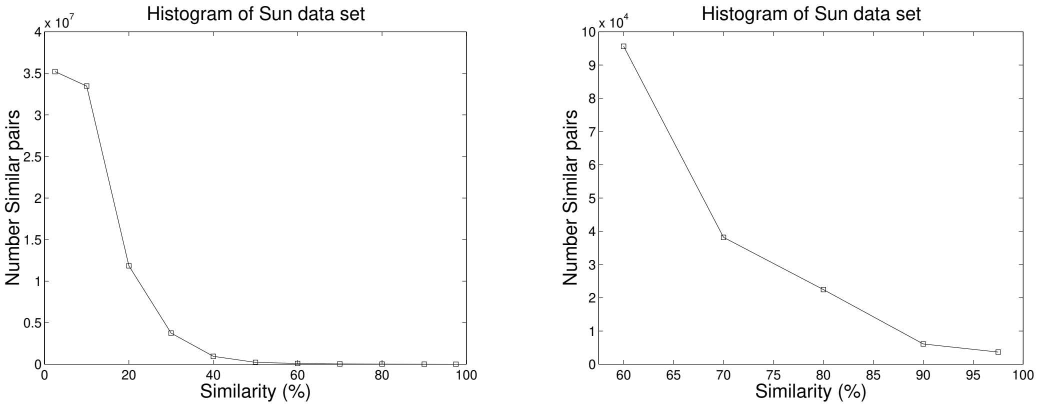


Figure 3: The first figures shows the similarity distribution of the Sun data. The second shows again the same distribution but it focuses on the region of similarities that we are interested in.

5 Experiments

We have conducted experiments to evaluate the performance of the different algorithms. In this section we report the results for the different experiments. We use two sets of data namely synthetic data and real data.

Synthetic Data: The data contains 10^4 columns and the number of rows vary from 10^4 to 10^6. The column densities vary from 1% to 5% and for every 100 columns we have a pair of similar columns. We have 20 pairs similar columns whose similarity fall in the ranges (85, 95), (75, 85), (65, 75), (55, 65) and (45, 55).

Real Data: The real data set consists of the log of HTTP requests made over a period of 9 days to the Sun Microsystems Web server (`www.sun.com`). The columns in this case are the URL's and the rows represent distinct client IP addresses that have recently accessed the server. An entry is set to 1 if there has been at least one hit for that URL from that particular client IP. The data set has about thirteen thousand columns and more than 0.2 million rows. Most of the columns are sparse and have density less than 0.01%. The histogram in Figure 3 shows the number of column pairs for different values of similarity. Typical examples of similar columns that we extracted from this data were URLs corresponding to `gif` images or `Java` applets which are loaded automatically when a client IP accesses a parent URL.

To compare our algorithms with existing techniques, we implemented and executed the a-priori algorithm [1, 2]. We would like to mention that although a-priori is not designed for this task it is the only existing technique and gives us a benchmark to evaluate our algorithms. The comparison

Support threshold	Number of columns after support pruning	A-priori (sec)	MH (sec)	K-MH (sec)	H-LSH (sec)	M-LSH (sec)
0.01%	15559	-	71.4	87.6	15.6	10.7
0.015%	11568	96.05	44.8	52.0	6.7	9.7
0.2%	9518	79.94	25.8	36.0	6.0	5.1

Figure 4: Running times for the news articles data set

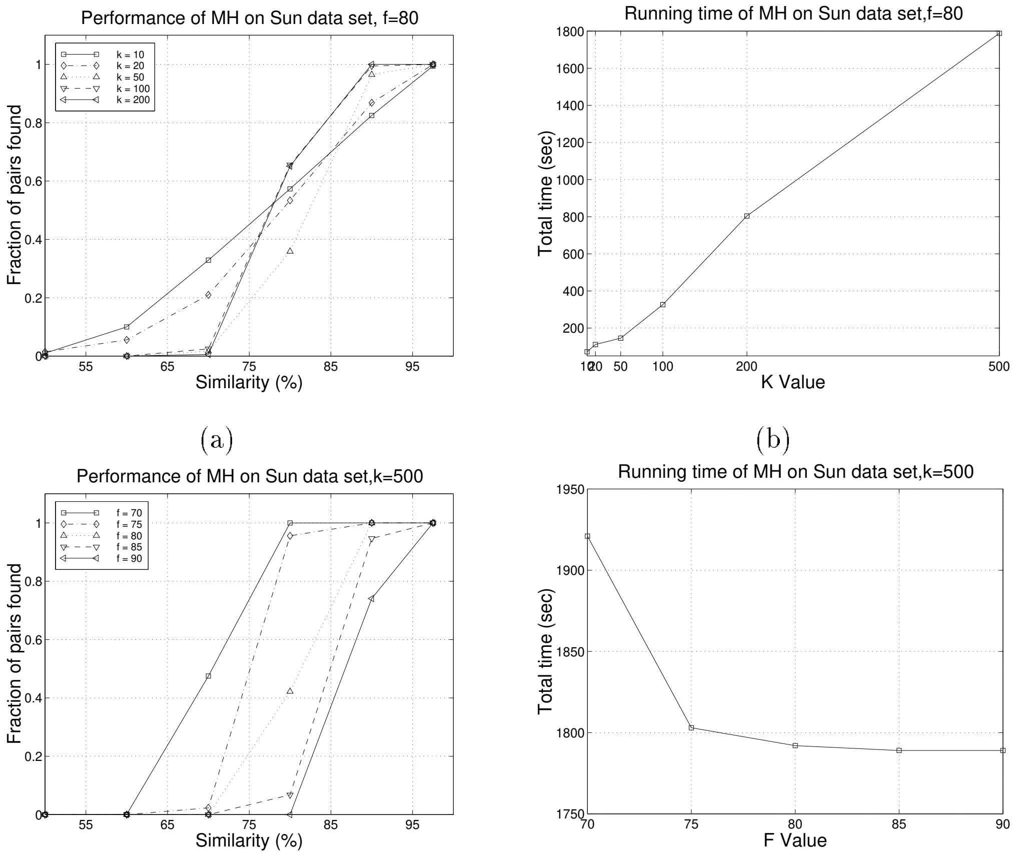
was done for the news articles data that we have mentioned in section 2.

We conducted experiments on the news article data and our results are summarized in Figure 4. The a-priori algorithm cannot be run on the original data since it runs out of memory. Hence we do support pruning to remove columns that have very few ones in them. Each pruned data stands for a row of the table and the comparative results for each data are reported in the row. It should be noted that for support threshold of 0.01% and less, a-priori algorithm runs out of memory on our systems and does a lot of thrashing. It should be noted that although our algorithms are probabilistic they report the same set of pairs as that reported by a-priori.

5.1 Results

We implemented the four algorithms described in the previous section, namely MH, K-MH, H-LSH, and M-LSH. All algorithms were compared in terms of the running time and the quality of the output. Due to the lack of space we report experiments and give graphs for the Sun data, which in any case are more interesting, but we have also performed tests for the synthetic data, and all algorithms behave similarly.

The quality of the output is measured in terms of false positives and false negatives generated by each algorithm. To do that, we plot a curve that shows the ratio of the number of pairs found by the algorithm over the real number of pairs for a given similarity range (e.g. Figure 8). The resulting plot is typically an "S"-shaped curve, that gives a good visual picture for the false positives and negatives of the algorithm. Intuitively, the area below the curve and left to a given similarity cutoff corresponds to the number of false positives, while the area above the curve and right to the cutoff corresponds to the number of false negatives. The real number of pairs within a similarity range was computed in an off-line fashion by a brute-force counting algorithm, and used for comparison



(a) (b)

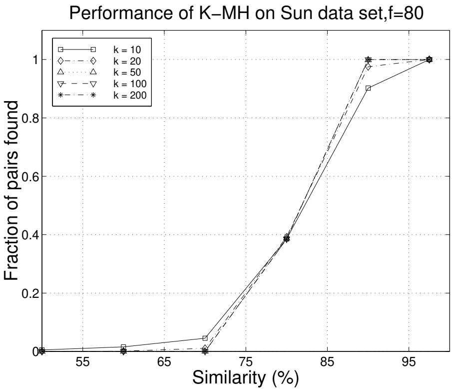
(c) (d)

Figure 5: Quality of output and total running time for MH algorithm as k and s are varied

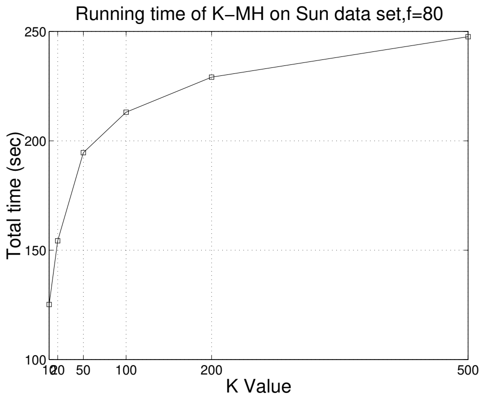
in all of our experiments. It was feasible in our case because the number of columns in our real data was small enough to permit keeping counters for all pairs in the main memory. But one could also achieve it by making multiple passes over the data. However, this is clearly not a scalable approach.

We now describe the behavior of each algorithm as their parameters are varied.

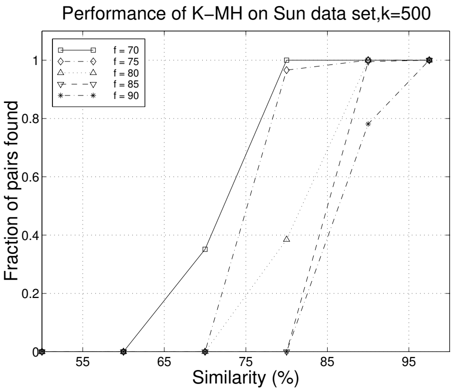
MH and K-MH algorithms have two parameters, s^* the user specified similarity cutoff, and k, the number of Min-Hash values extracted to represent the signature of of each column. Figures 5(a) and 6(a) plots "S"-curves for different values of k for the MH and K-MH algorithms. As the k value increases the curve gets sharper indicating better quality. In figures 5(c) and 6(c) we keep k fixed and change the value s^* of the similarity cutoff. As expected the curves shift to the right as the cutoff value increases. Figures 5(d) and 6(d) show that for a given value of k the total running time decreases marginally since we generate fewer candidates. Figure 5(b) shows that the



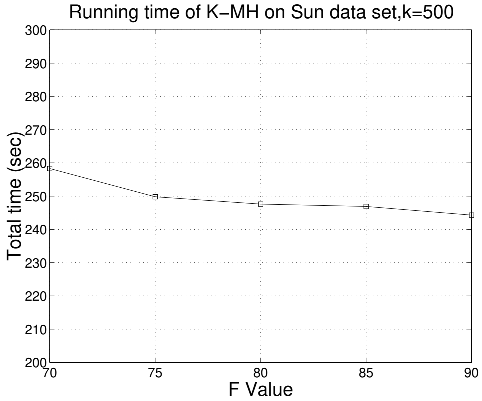
(a)



(b)



(c)



(d)

Figure 6: Quality of output and total running time for K-MH algorithm as k and s are varied

total running time for MH algorithm increases linearly with k. However this is not the case for K-MH algorithm as depicted by figure 6(b). The sub-linear increase of the running time is due to the sparsity of the data. More specifically, the number of hash values extracted from each column is upper bounded by the number of ones of that column, and therefore, the hash values extracted do not increase linearly with k.

We do a similar exploration of the parameter space for the M-LSH and H-LSH algorithms. The parameters of this algorithm are r, and l. Figure 8(a) and 7(a) illustrates the fact that as r increases the probability that columns mapped to the same bucket decreases, and therefore the number of false positives decreases but as a trade-off consequence the number of false negatives increases. On the other hand, figure 8(c) and 7(c) shows that an increase in l, corresponds to an increase of the collision probability, and therefore the number of false negatives decrease but the number of false positives increases. Figures 8(b) and 7(b) show that the total running time increases with l since

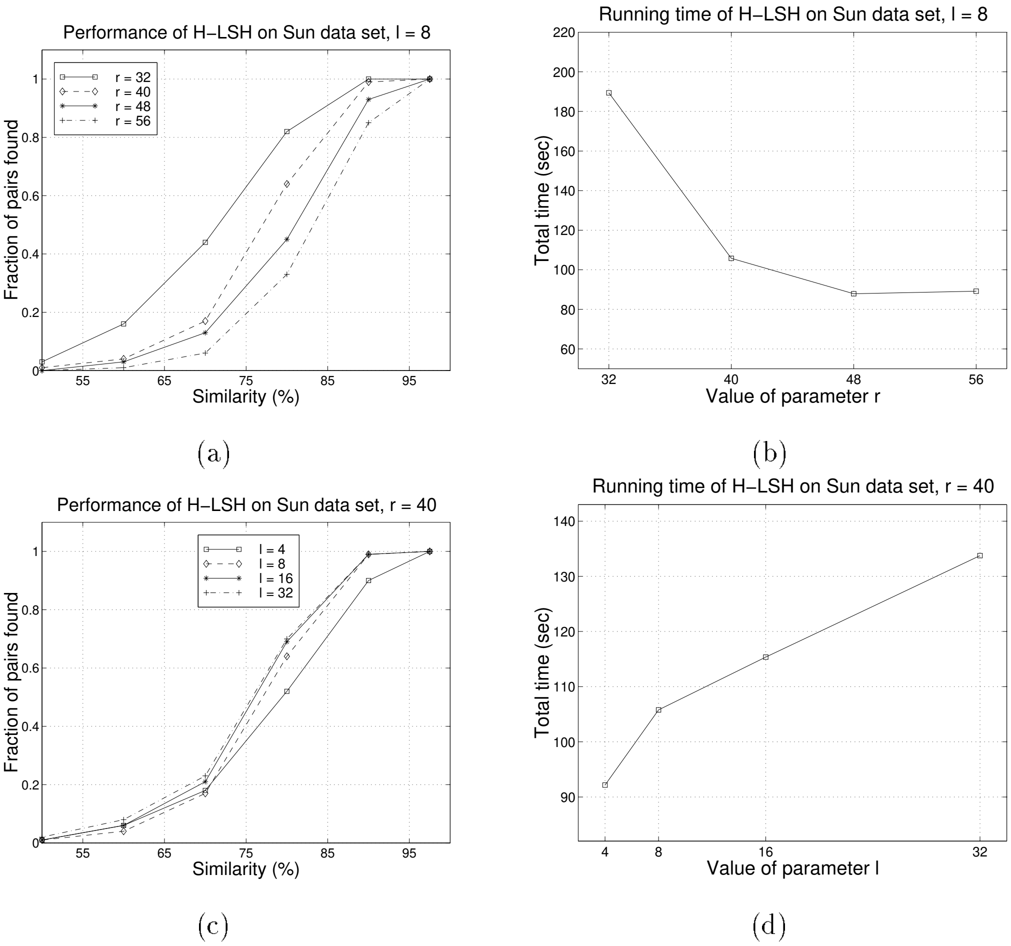
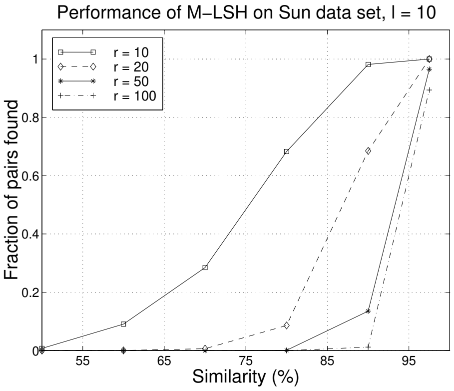


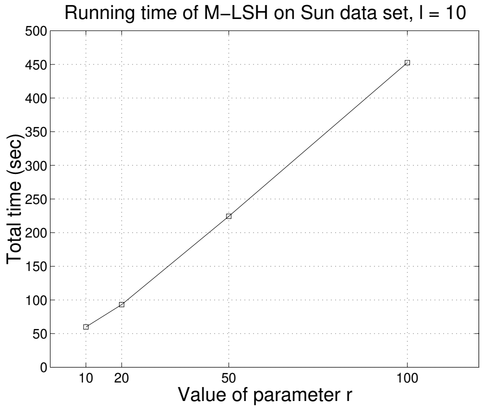
Figure 7: Quality of output and total running time for H-LSH algorithm as r and l are varied

we hash each column more times and this also results in an increase in the number of candidates. In our implementation of M-LSH, the extraction of min hash values dominates the total computation time, which increases linearly with the value of r. This is showed in figure 8(c). On the other hand, in the implementation of H-LSH, checking for candidates dominates the running times, and as a result the total running time decreases as r increases since less candidates are produced. This is showed in figure 7(c).

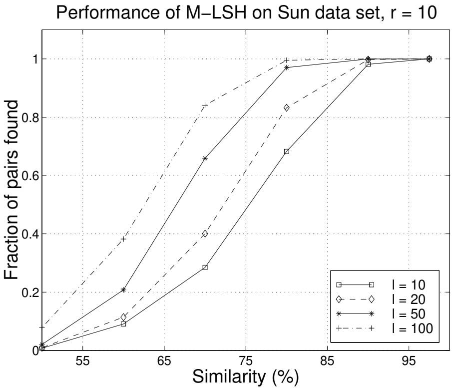
We now compare the different algorithms that we have implemented. When comparing the time requirements of the algorithm we compare the CPU time for each algorithm since the time spent in I/O is same for all the algorithms. It is important to note that the for all the algorithms the number of false negatives is very important and this is the quantity that requires to be kept in control. As long as the number of false positives is not too large (i.e. all of candidates can fit in main memory) we can always eliminate them in the pruning phase. To compare the algorithms we



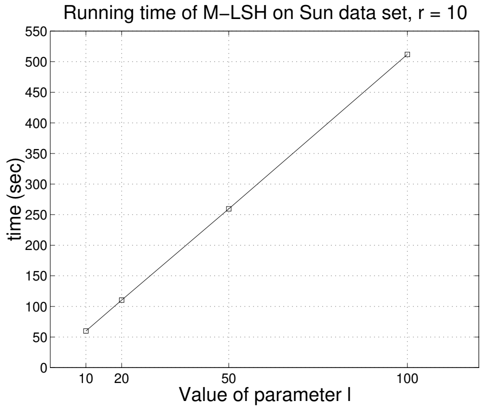
(a)



(b)



(c)



(d)

Figure 8: Quality of output and total running time for M-LSH algorithm as r and l are varied

fix the percentage of false negatives that can be tolerated. For each algorithm we pick the set of parameters for which the number of false negatives is within this threshold and the total running time is minimum. We then plot the total running time and the number of false positives against the false negative threshold.

Consider Figures 9(a) and 9(c). The figures shows the total running time against the false negative threshold. We can see that the H-LSH algorithm requires a lot of time if the false negative threshold is less while it does better if the limit is high. In general the M-LSH and H-LSH algorithms do better than the MH and K-MH algorithms. However it should be noted that H-LSH algorithm cannot be used if we are interested in similarity cutoffs that are low. The graph shows that the best performance is shown by the M-LSH algorithm.

Figure 9 gives the number of false positives generated by the algorithms against the tolerance limit. The false positives are plotted on a logarithmic scale. In case of H-LSH and M-LSH algorithms the number of false positives decreases if we are ready to tolerate more false negatives since in that case we hash every column fewer times. However the false positive graph for K-MH and MH is not monotonic. There exists a tradeoff in the time spent in the candidate generation stage and the pruning stage. To maintain the number of false negatives less than the given threshold we could either increase k and spend more time in the candidate generation stage or else decrease the similarity cutoff s and spend more time in the pruning stage as we get more false positives. Hence the points on the graph correspond to different values of similarity cutoff s^* with which the algorithms are run to get candidates with similarity above a certain threshold. As a result we do not observe a monotonic behavior in case of these algorithms.

We would like to comment that the results provided should be analyzed with caution. The reader should note that whenever we refer to time we refer to only the CPU time and we expect I/O time to dominate in the signature generation phase and pruning phase. If we are aware about the nature of the data then we can be smart in our choice of algorithms. For instance the K-MH algorithm should be used instead of MH for sparse data sets since it takes advantage of sparsity.

6 Extension to High-Confidence Association Rules

So far we have devised techniques for finding column pairs that have high "similarity". However one would be interested in finding column pairs that have high confidence measure. These are Association rules without the support requirement. This was the problem that motivated our work as explained earlier. We now look at how our techniques can be extended to finding such rules.

The confidence of the rule $c_i \Rightarrow c_j$ denoted by $conf(c_i, c_j)$ is given by $conf(c_i, c_j) = |C_i \cap C_j|/|C_i| = S(c_i, c_j)|C_i \cup C_j|/|C_i|$. Consider the Min-Hash Schemes. We obtain the summary matrix $\widehat{M}$ and use it to estimate the value of $S(c_i, c_j)$ by counting the fraction of rows on which the two columns agree. If the matrix is sufficiently large (as seen earlier in Theorem 1) then we get a good estimate of $S(c_i, c_j)$. It remains to calculate or estimate the value $|C_i|/|C_i \cup C_j|$. It can be easily seen[2] that $Pr(h(c_i) \leq h(c_j)) = |C_i|/|C_i \cup C_j|$. Thus in order to estimate this quantity we should also get the fraction of rows for which the hash values of c_i are no greater than that of c_j. The Row-Sorting technique can be extended to do this. We maintain two sets of counters for each column c_i: one for counting the number of rows for which each column c_j agrees with the hash value of c_i and the other for counting the number of rows for which the hash value of c_j is at least

[2]the proof is identical to the proof of $Pr(h(c_i) = h(c_j)) = S(c_i, c_j)$.

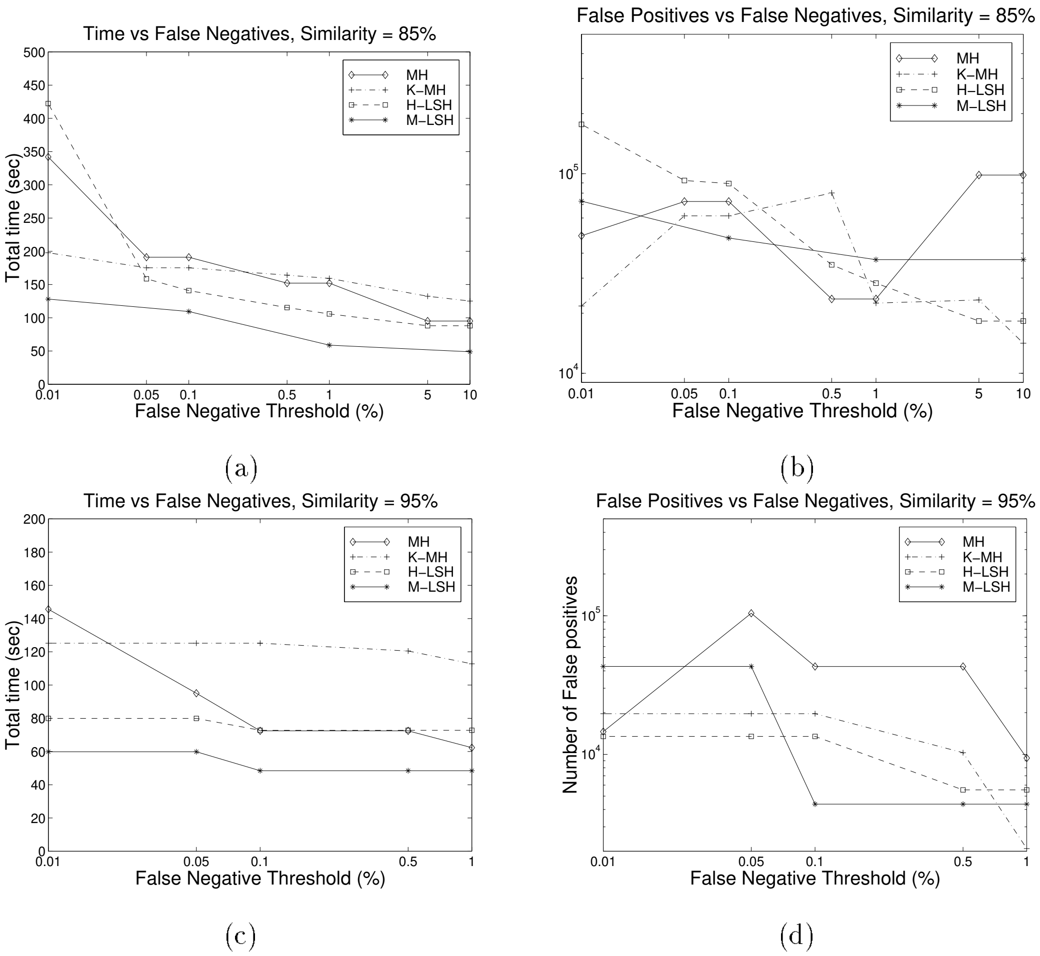


Figure 9: Comparison of different algorithms in terms of total running time and number of false positives for different negative thresholds.

as much as that of c_i. The time required to do this is $O(km^2)$. However the value of $S(c_i, c_j)$ as well as $|C_i|/|C_i \cup C_j|$ may be small. As a result we may require a bigger table $\widehat{M}$ to accurately estimate the two. Note that we cannot use the Hash-Count technique to do this. For similar reasons the Min-LSH scheme also cannot be used for this purpose.

We suggest an alternate technique if we are looking for very high confidence column pairs. One should note that $S(c_i, c_j)$ is a lower bound for $conf(c_i, c_j)$ as well as $conf(c_j, c_i)$. Thus we can surely declare as candidates column pairs for which the estimate of $S(c_i, c_j)$ exceeds the confidence threshold. Moreover it is easy to see that if $conf(c_i, c_j) \approx 1$ then $S(c_i, c_j) \approx |C_i|/|C_j|$. Thus we can declare as candidates those column pairs for which $S(c_i, c_j) \approx |C_i|/|C_j|$. The experimental results are qualitatively the same as for similar column-pairs.

7 More Extensions and Further Work

We briefly discuss some other extensions of the results presented here as well as directions for future work.

We can use our Min-Hashing scheme to determine more complex relationships, e.g., c_i is highly-similar to $c_j \vee c_{j'}$, since the hash values for the induced column $c_j \vee c_{j'}$ can be easily computed by taking the component-wise minimum of the hash value signature for c_j and $c_{j'}$. Extending to $c_j \wedge c_{j'}$ is more difficult. It works as follows. First, observe that "c_i implies $c_j \wedge c_{j'}$" means that "c_i implies c_j" and "c_i implies $c_{j'}$". The latter two implications can be generated as above. Now, we can conclude that "c_i implies $c_j \wedge c_{j'}$" if (and only if) the cardinality of c_i is roughly that of $c_j \wedge c_{j'}$. This presents problems when the cardinality of c_i is really small, but is not so difficult otherwise. The case of small c_i may not be very interesting anyway, since it is difficult to associate any statistical significance to the similarity in that case. It is also possible to define "anti-correlation," or mutual exclusion between a pair of columns. However, for statistical validity, this would require imposing a support requirement, since extremely sparse columns are likely to be mutually exclusive by sheer chance. It is interesting to note that our hashing techniques can be extended to deal with this situation, unlike a-priori which will not be effective even with support requirements. Extensions to more than three columns and complex boolean expressions are possible but will suffer from an exponential overhead in the number of columns.

8 Summary

We now present a summary of the problem and the various algorithmic techniques described in this paper. Given a relation R containing n tuples over a set of *boolean* attributes $A_1, A_2, \ldots, A_m$ in the form of a $m \times n$ *boolean* matrix. The goal is to find all pairs of columns (c_i, c_j) whose similarity $S(c_i, c_j)$ as defined in Section 1 is greater than an user specified threshold s^*.

The problem can be easily solved if the data fits in main memory. We address the case when data size is large. We have presented four techniques to solve this problem. All these techniques are probabilistic and generate a set of candidates. We propose that these candidates be verified to eliminate any false positives. We now briefly summarize the candidate generation for each of these techniques.

The first technique called Min-Hashing (MH) generates a signature for each column which is a set of K Min-Hash values. These signatures are stored in the main memory and for each column pair we count the number of Min-Hash values they agree on. If they agree on at least a certain fraction the pair is flagged as a candidate. The probability that a Min-Hash value for two columns is the same is equal to the similarity between them. As a result this technique allows us to generate a "summary" of the data which we can use to generate the candidates. However, the technique has two drawbacks: It requires us to generate k Min-Hash values for each column in order to get accurate candidates which increases the running time since the generation of the "summary" is linear in the number of Min-Hash values. Secondly the generation of candidates using the summary is quadratic in the number of columns in the worst case.

In order to address the first drawback we introduced the second technique called the K Min-Hashing (K-MH) which lets us generate the signature for each column without computing k different Min-Hash values. This slightly compromises quality of the output but helps in reducing the running time. The way we do this is to use the k smallest Min Hash values for each column instead of using just the minimum and repeating it k times.

To address the second drawback we introduced the technique called Min-LSH (M-LSH). Each column is hashed into a bucket using the concatenation of a few Min-Hash values as its hashing key. This is repeated few times. If two columns agree on a certain fraction of the Min-Hash values then with high probability they will fall into the same bucket at least once. All columns that hash into the same bucket are pairwise declared candidates. This avoids the quadratic (in the number of columns) running time in candidate generation phase and is proven to be very effective in reducing the running time. Our results show that this technique gives very good results.

The last technique called the Hamming-LSH (H-LSH) is different from the others in that it does not generate signature using Min-Hash values but instead works directly on the data. If all the columns had the same density then the similarity between two columns is related to the hamming distance between the two. Smaller the hamming distance greater is the similarity. We use this intuition and hash the columns with similar density such that columns with small hamming distance fall into the same bucket. As before all columns falling into the same bucket are declared candidates. This technique is effective if we are looking for high similarity cutoffs and if we are ready to accept a few false negatives.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. "Mining Association Rules Between Sets of Items in Large Databases". In *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1993, pp. 207–216.

[2] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In *Proceedings of the 20th International Conference on Very Large Databases,* 1994.

[3] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data". In *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1997, pp. 255–264.

[4] A. Broder. "On the resemblance and containment of documents". In *Compression and Complexity of Sequences (SEQUENCES'97)*, 1998, pp. 21–29.

[5] E. Cohen. "Size-Estimation Framework with Applications to Transitive Closure and Reachability". *Journal of Computer and System Sciences* 55 (1997): 441–453.

[6] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* A Wiley-Inter science Publication, New York, 1973.

[7] A. Gionis, P. Indyk, and R. Motwani. "Similarity Search in High Dimensions via Hashing". In *Proceedings of 25th International Conference on Very Large Data Bases,* 1999, pp. 518–529.

[8] D. Goldberg, D. Nichols, B.M. Oki, and D. Terry. "Using collaborative filtering to weave an information tapestry". *Communications of the ACM* 55 (1991): 1–19.

[9] S. Guha, R. Rastogi, and K. Shim. "CURE - An Efficient Clustering Algorithm for Large Databases". In *Proceedings of the ACM-SIGMOD International Conference on Management of Data,* 1998, pp. 73–84.

[10] J.M. Hellerstein, P.J. Haas, and H.J. Wang. "Online Aggregation". In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1997.

[11] P. Indyk and R. Motwani. "Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality". In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing,* 1998, pp. 604–613.

[12] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[13] N. Shivakumar and H. Garcia-Molina. "Building a Scalable and Accurate Copy Detection Mechanism". In *Proceedings of the 3rd International Conference on the Theory and Practice of Digital Libraries*, 1996.

[14] C. Silverstein, S. Brin, and R. Motwani. "Beyond Market Baskets: Generalizing Association Rules to Dependence Rules". Preliminary version: In *Proceedings of the ACM SIGMOD Conference on Management of Data,* 1997, pp. 265–276. Journal version: *Data Mining and Knowledge Discovery* 2 (1998): 69–96.

[15] C. Silverstein, S. Brin, R. Motwani, and J.D. Ullman. "Scalable Techniques for Mining Causal Structures". In *Proceedings of the 24th International Conference on Very Large Data Bases*, 1998, pp. 594–605.

[16] H.R. Varian and P. Resnick, Eds. CACM Special Issue on Recommender Systems. *Communications of the ACM* 40 (1997).