Efficient Computation

Sampling implicit sets: a new data mining technique

Ph.D. Dissertation
Presented to the Faculty of the IT University of Copenhagen
in Partial Fulfilment of the Requirements for the Ph.D. Degree

Candidate
Andrea Campagna

Supervisor
Rasmus Pagh

Assessment Committee:
Thore Husfeldt (Chair)
IT Universitetet i København, Denmark
Lund University, Sweden

Gerth S. Brodal
Aarhus University, Denmark

Piotr Indyk
Massachusetts Institute of Technology, USA

Year 2011

Acknowledgements

I would like to thank Professor Rasmus Pagh for these years of research during which he has been a precious guide. If I have gained any skill as a researcher during this period in time, it is because of his lessons and his advices.
A special thank goes to Professor Gerth Brodal, Professor Thore Husfeldt and Professor Piotr Indyk for having accepted the burden of being in my assessment committee.
I thank Professor Ronitt Rubinfeld for having been a very nice host and for the great support that she has given me in order to deal with a completely new research topic.

A thank goes to the *Mitteleuropa* for the nice people, maybe friends. A special thought goes to the one person who, in these years, made a better being of me and then killed me. I fear I did the same to you.

Finally, my mind turns towards those who have given me the strength and resources to walk all the path to this point. In my brain I have both the ones who are still there and the ones who consumed. Without you it would have been impossible for me to make this walk, for several and distinct reasons.
The cliffs I will climb, the peaks I will attack: and the sky.
The track I will live, the consequences I will take: and the pain.

A Karl Marx

L'homme,
étant condamné à être libre,
porte le poids du monde
tout entier sur ses épaules.

Abstract

One of the most interesting new problems in theoretical computer science is massive data algorithmics. Massive data problems cannot rely on naïve techniques: Even quadratic time algorithms can turn out to be unbearable in the case of massive input. This problem is even more urgent when the output depends on aggregations of the input data, since to aggregate data translates in an explosion in size and time necessary to compute the aggregation. Still, the aggregate, intermediate data, are necessary to produce the correct output. We present algorithms that are able to sample from the multiset of intermediate data, without representing explicitly such a multiset, in this way achieving small space usage and efficient running time.

We address problems in data mining, data streaming, pattern mining in graphs and structure prediction. For all these topics, we use the innovative technique of sampling from implicit sets.

More specifically, we present algorithms for finding the most similar pairs of items in the so called market basket model. This model is easily explained in the following way: we want to find those pairs of items that are mainly bought together by customers of a shop. In one scenario, the input of this problem is given in a single unit and can be read several times; on the other hand, there are frameworks in which the input arrives in smaller, volatile chunks; in these frameworks it is infeasible to store more than one chunk at a time; therefore, once a chunk is read, it cannot be accessed anymore in the future. We address the problem of finding similar pairs in both scenarios without generating the multiset of all the pairs that appear in the customers' baskets. For the latter scenario we also present two hardness results.

Furthermore, we present an algorithm for finding recurrent sequences of labels in a directed graph where vertices are, non uniquely, labelled. The output is produced without generating all the possible directed paths of nodes that exist in the graph. To the best of our knowledge, this is the first algorithm dealing with such a problem.

Moreover, we show an algorithm for computing an approximation of the number of non zero entries in the result of the product of two boolean matrices. The estimate is output without explicitly producing the result of the matrix product.

The algorithms we present are all randomised, and so they can make errors. A thorough analysis of these errors shows that our algorithms are indeed accurate and reliable, and errors are unlikely. Moreover, all our algorithms solve the respective problems using time, in expectation, that is linear, or quasi linear, with respect to the input size or is linear with respect to the size of the output.

Contents

Chapter 1

Introduction

Theoretical computer science has evolved in the years, starting from the classical graphs and combinatorial optimisation problems, to arrive to the arguably more recent massive data and algorithmic game theory problems.

This evolution accounts for many factors that influenced computer science during the last 50 years: Faster machines, cheaper storage space, pervasive deployment of Internet in the first world, economical evolution and a different perception of the tasks of scientific research in universities. While all these factors are not necessarily positive, the plethora of new branches that computer science has followed has highlighted some interesting and deep problems.

A very meaningful example concerning these new problems is the one connected to massive data algorithms. The topic has acquired such a huge relevance that entire research centres have been devoted to researching in this field. Why massive data algorithms have become a central topic of study is easily understood, looking at the evolution of western societies in the last 30 years. Besides the political reasons and interests of the police in keeping the data of social entities under strict control, cheap hardware has caused all the administrative tasks of companies or social institutions to be carried out using computers. All that was formerly recorded only on paper, suddenly found a Platonic copy of itself in a digital format. This means that managing such a huge amount of information could not rely anymore on naïve algorithms. Managing information is not only the task of accessing and storing the data; it also entails being able to aggregate them and output the result of the aggregation tasks. Handling information also requires to run analyses of the data in order to find some possibly interesting characteristics that they possess. Typically, companies want to build profiles of the customers, in order to define classes of clients on the basis of their habits. A very good idea of this process can be obtained by thinking of the use that of our personal information, such as the contents of unencrypted emails, our World Wide Web habits and the country from which we connect, is done by some of the largest companies operating on the Internet. So it is very common to get personalised adverts, references to digital information concerning people we might know in real life, and so on. Another factor that has made massive data algorithms important has been the fast development of the Internet on a world scale, and in particular the huge increase in data traffic that this phenomenon has created. Analysis of traffic implies facing an overwhelming amount of data. Just the log of a home router can easily grow to megabytes or even gigabytes in

size, hence the obvious considerations about ISPs and backbone routers. Analysis of those logs can be highly interesting for many actors in the digitalised world. In particular, such analyses have to be run on the fly, while the packets pass through the router, in order for this information to be of any use. This specific setting has also made a new problem model popular, that completely catches the issues of carrying on fast computations with a very limited amount of memory.

In this massive data world, this thesis is born and grown up, showing some characteristic of problems and proposing some solutions.

In what remains of the chapter, we will explain which problems we face and how they relate to each other. Moreover we will devote a large part of the chapter to introduce randomized algorithms and techniques.

1.1 A descriptive overview

Here we describe in a general, descriptive and abstract way, the problems we address. The section does not require the reader to have any scientific knowledge, with respect to computer science and mathematics.

Before starting the description of the problems, it is opportune to define what massive data means. Massive data problems face inputs that are overwhelmingly large given the computational power at hand. Terabytes of data are often used to represent some information; for example very long sequences of items, market basket data of huge multinational shops, the content of the World Wide Web, are all good representatives of massive data for the current technology.

However, we think it is relevant to highlight the fact that the meaning of massive has not to deal only with the amount of memory necessary to store the input. A 1000 nodes graph can be represented using a very small quantity of memory; still it is, and it will likely always be, a massive data input when the problem to solve is the Travelling Salesperson Problem[1]. This depends from the fact that a graph is *implicitly* massive. Its rich mathematical structure is able to represent in a succinct way a huge number of sub structures that are often the objects of interest for some problem to be solved. More information about the TSP and its state-of-the-art, including the largest solved instance so far, can be found in the book [10].

1.1.1 Data mining

A typical massive data task is to extract information from data. This is not only the task of querying a database about its content. It is also the process of building non structured information that lie hidden in the rough data. Data mining is a big umbrella including many distinct problems. One of the most interesting amongst these problems, can be described in the following abstract form: Suppose that a set of people possesses collections of objects, one set per person. Moreover, suppose that all the objects in the collections come from a limited number of possible existing object kinds, so that several people can

[1] This problem, often called TSP, is very hard to solve. Suppose that the salesperson can sell his products in a certain number of cities that are connected by a network of roads; the salesperson, in order to maximise his revenue, wants to find a tour that touches all the cities only once and that allows him to use at most a fixed amount of fuel.

own objects of the same kind. A reasonable question is to ask which subsets of objects are often owned by people, or to ask which objects are likely to be owned together. In general, given a dataset, it is useful to understand which are the patterns, if any, that happen in the dataset. Moreover, once a pattern is discovered, it can be interesting to understand how frequent such a pattern actually is.

Example 1.1.1. We live in an industrialised post-modern society, so the request for cars with some very, so to say, advanced characteristics is very high. We are the car builders and we want to understand which have been the most well received features of the cars we produce by the market. In this setting, each subset is a car with a given set of characteristics. So a car can have a certain colour, a compact disk reader, the ABS braking system and so on. The goal of the mining would be finding those sets of characteristics that are highly related. In particular, looking at the cars sold in the previous years, it can happen to spot that red cars are nearly always equipped with black seats, and the other way around. Such data suggest that red frames and black seats are related characteristics. ◦

It is also interesting to point out that data mining not only attempts to uncover information that is spread, hidden and often unknown, in the data, but it tries as well to predict the evolution of the data, using the information that it is able to get. Looking at a pattern mining framework, where recurring patterns in the input are sought, it is easy to see how and why this works: Suppose that there is evidence that, when a series of patterns happens, then a consequence, a specific datum or data pattern, shows up. This means that when the series of patterns is spotted, it can be considered as likely that the consequence that has been discovered will happen as well. In the literature the two described approaches fall respectively in the so-called descriptive and predictive class of data mining. The latter is also a very common and studied problem in the framework of *machine learning*.

Example 1.1.2. In the past years, white cars with a pack rack and towing attachment have turned out to be quite popular, so we can infer that that set of characteristics has to be put on the market again, since it is likely to find a receptive audience. ◦

1.1.2 Streaming

Streaming algorithms are probably one of the main topics of interest in recent theoretical computer science. The setting in which a router, in an online fashion, tries to compute information as the packets flow through it, is entirely caught by the streaming framework and constitutes a reasonable real world explanation of what the streaming environment is. It is clear what are the issues that such algorithms have to face: Limited space, which results in the infeasibility of storing the whole input, and limited time, because of the high rate of the incoming data. The input of these algorithms is typically an infinite stream of data. An infinite stream of data is massive without any doubt and makes entirely clear the reason why storing the input is not an option.

Example 1.1.3. The car factory wants to differentiate its offer on the basis of territorial tastes and preferences. To achieve this goal, sensors along some of the highways are placed that can register the features of cars that pass by them. These devices are limited in computational power and storage space. For example they might be powered only with solar panels, so they need to save as much energy as possible. ◦

Many data mining tasks can be framed in a streaming fashion. It is easy to understand, as a matter of fact, that the information contained in a stream of data can be found using concepts and ideas from the data mining world. Of course the techniques used in the streaming world and the data mining one cannot be exactly the same, given the difference in nature of the two problems, but can be adapted sometimes to fit each other framework.

An interesting challenge in streaming, as well as in data mining, is exploiting any possibility of speeding up the computation using parallelism. Nowadays even laptops are equipped with multicore CPUs, so parallel algorithms can provide attractive solutions when the data are massive and the time constraints are strict.

Example 1.1.4. Instead of processing the data arriving from the sensors like if they form a single stream, we split those data in several streams, on the basis of the areas where the sensors are placed. In this way we end up with several streams of approximately the same size, since all highways are used with the same average intensity. Hence we can process each stream using a distinct CPU and reduce the time needed to find the information we are interested in. ◦

1.1.3 Graph mining

Graphs are notoriously a natural source of complex problems. Their structure embeds a level of mathematical complexity and richness that often challenges algorithms that want to be efficient.

Data mining often asks for mining over some very specific data coming from real world applications. A worthy goal is thus to come out with general approaches that are able to capture the structure of the problem. In this way it is often possible to find algorithms that are able to solve a much wider array of problems that are representable with the formal, as said, general, formulation of the problem.

Graphs, as noted before, are very rich in structure and can productively be applied to some of the problems that data mining poses. Graph mining often looks for recurring patterns in a graph. Often, these patterns are substructures of the graph that become interesting when they repeat frequently.

As an example, consider a situation in which a certain number of agents have installed transmitters on themselves and there are antennas in their environment, that are capable of keeping track of the passage of an agent when it is reasonably close. Along with the event of having seen a specific agent, the antenna keeps track of the timestamp of the reading. In such a setting it can be interesting to spot those patterns of movement that agents tend to exhibit in a temporal sequence, considering two antennas' readings consecutive if they

happen within a certain time window. To be more concrete, consider people pushing around trolleys in a mall. Suppose that there is a transmitter installed on every trolley. It can be commercially advantageous to keep track of what are the recurrent and frequent movements of customers in the mall, such as which shops people visit in a sequence. This information might allow a market analyst to be able, in real time, to predict the movement of people, according to patterns exhibited by other customers.

These settings can be represented using a graph, in particular, a directed acyclic graph, where events are vertices of the graph and edges represent the fact that two events happened within a reasonable time window. In general, when a real-world situation can be represented using a directed graph, so that two events can be considered close according to a certain metric, which needs not be the time, finding frequent patterns in such a graph allows mining the sequences of events that are frequent. An example of this flexibility is the one that follows.

Example 1.1.5. In our world of massive industrial production, the sensors that were placed on the highways have been re-converted to just signal the presence of a car, registering the plate of the car and the time when it passed. We might be interested in computing which routes are the most common that cars take. This would help building a network of dedicated workshops, for instance, to assist owners of cars experiencing mechanical problems. The whole road system can be represented with a directed graph. In this setting, car routes can be represented using directed sequences of nodes in the directed graph, so the task of the graph mining algorithm would be to find those routes, so those directed paths, occurring frequently. ◦

1.1.4 Matrix multiplication

Using a matrix many information and characteristics of datasets can be represented. The idea is that each row or column of a matrix can be seen as the representation of a vector. Since objects can be represented with a list of values in connection with the characteristics they provide, hence a vector, matrices turn out to be good tools for representing sets of objects.

Example 1.1.6. A car can be described using a set of characteristics. If we fix an order for this set of characteristics, we can associate a vector with each car, that is a sequence of numbers, such that the position of a number in the sequence tells us what characteristic it refers to, and the value quantifies the characteristic. A red car, with 5 doors, no ABS and three air bags may be represented as $(1, 5, 0, 3)$. We are assuming that to each colour is represented uniquely by a number, so to red, the number 1 is associated. If we list all the cars, their associated vectors will form a matrix. ◦

Matrix multiplication is therefore a very interesting abstraction of many problems in the massive data environment, when the input needs to be manipulated in order to get the structure necessary to uncover the information that it embeds. Matrix multiplication essentially captures, amongst others, the concepts of set intersection and its size, cardinality of sets *tout court*. These

concepts and mathematical operations are very important in database applications. In this field it is often interesting to know how many elements an entity possesses, which translates, in practical terms, to removing duplicates from the result of some relational algebra operators.

Moreover, matrix multiplication is interesting as a problem in its own right, and the research community has used a lot of effort in solving it in an efficient way. The mathematical structure of the problem is rich enough that we can hope to find some aspects that have not yet been explored.

Example 1.1.7. We associate a vector of characteristics to each car. For concurrency reasons, we want to compute the similarity amongst two kinds of cars on the basis of the gadgets they provide. This will allow us to differentiate our products with respect to other producer and within the own range of cars offered. This problem is very easily represented by multiplying the characteristic vectors of distinct models of cars. ◦

1.2 This thesis

As introduced in the previous sections, massive data require fast and space efficient algorithms. Still, it does not always suffice to just carry on the computation on the data in the form they are given. Often, the information is contained in some aggregation of the data. It is the same as in industrial products, where starting from the raw materials, a finished product comes out. This does not happen in a single step of production, the process usually goes through several phases of manipulation, where several semiprocessed products are produced. Often, the interesting information has to be found and extracted from these semiprocessed products rather than from the raw material.

A semiprocessed product often occupies much more space than the raw materials used for producing the semiprocessed product itself. As an example, one can think of car production, where the raw materials are metal and raw chemicals for plastic, and a semiprocessed product is the car frame and some of the plastic interiors. Moreover, producing those pieces is not an immediate task and, before they are actually ready to be put on the assembly line to get the car finished, a lot of time is necessary.

All this easily translates in terms of algorithms and complexity. Massive data are, tautologically, massive, so it is already challenging to deal with them in a space and time efficient way. When the data in their raw form are not sufficient to produce the output and they instead need to be aggregated, or semiprocessed to bind the idea to the former paragraph scenario, the explosion in size might be unbearable. This means that not all the semiprocessed products must be produced, because it would be just too expensive, both in terms of time and space.

For this reason this work has the title “Sampling implicit sets: A new data mining technique”: the sampling, which is one of the necessary techniques for dealing with massive datasets, happens on the implicit stock of semiprocessed products, without actually producing them all. As a concrete example we can think of customers who, in the context of market research, have expressed a list of characteristics that they would like to find in a car. Then, instead of

producing cars with all possible combination of characteristics, such as colour of the frame and electric engine, we produce cars having some of the combinations of characteristics: The ones that customers have expressed as important to have together. Notice that the sentence "important to have together" requires, asks for a proper semantic

For all the informally described problems, we use the technique of computing the output without producing what would be an overwhelming amount of data.

1.3 Our results

We now describe in a more precise way the problems we address, providing some terminology and defining some concepts and functions that will be used in the following chapters. These definitions will also be given in the appropriate chapters, but they are necessary at this point in order to have a mathematical idea of the problems we will present and the tools to provide a description of how they relate together.

The reader of the section is supposed to have some basic knowledge of computer science and theory of computing, since many definitions are skipped and concepts are assumed to be well known.

Each following sub-section will detail what has been described in general in Section 1.1.

1.3.1 Similarity mining - a two passes approach

We explain here what has already been described in an informal fashion in Section 1.1.1.

The input of this problem is a collection of subsets $T_1, \ldots, T_m$ of a set, or universe, $U = [n] = \{1, \ldots, n\}$. The average size of the sets T_i in the collection is represented by b. The letter T for the elements of the collection stands for *transaction*, which is a term borrowed from the world of databases. The problem is to produce those $2-$itemsets of elements of U that are highly similar, according to some similarity measure. It is evident how this problem can be trivially solved, producing the set $\binom{U}{2}$ of all the $2-$itemsets of U and studying its composition. It is evident, as well, that this approach, when the cardinality n of U is large, is not feasible, since it would produce $\binom{n}{2}$ pairs. Still, the implicit set from which we will sample, is exactly $\binom{U}{2}$. It is opportune to remark that $|\binom{U}{2}| = O(n^2)$. The main idea we will exploit is to spend, for a pair, a time that is proportional to the similarity of the pair itself. In particular this will translate in sampling a pair a number of time that is, in expectation, proportional to the similarity it holds. Relying on this fact, it will be possible to select, in a successive phase, those pairs that have been sampled frequently enough, since those will be the pairs that have a high similarity.

A more precise definition of similarity will be provided in the chapter that describes the algorithm. Here, it suffices to say that the similarity functions that we will address, depend linearly on the number of occurrences of a pair and in inverse proportion to the number of occurrences of the single items the pair contains. Just as an example, we can think of a similarity measure that is sometimes actually used, called *support*. *Support* measures the number of occurrences of a pair, so the more frequently a pair appears in the input, the

more similar it is considered, according to this measure. A technique in this case would be to sample uniformly from the transactions. The sampling probability should be chosen as a function of the minimum support Δ we are interested in. In particular, we can choose a sampling probability $\rho = \mu/\Delta$, so that a pair whose support is equal to Δ will be sampled μ times. Moreover, pairs having support much smaller than Δ would be unlikely to be sampled even once. Besides the time necessary for reading the input, using a smart way of carrying out the sampling, we could spend time only on pairs that are actually sampled. In this way, we would obtain a random sample of the pairs such that a pair appears in the sample a number of times, in expectation, proportional to its similarity, that is, its support s times the sampling probability ρ. This would give us a good idea of the high frequency pairs. Notice that *false positives*, pairs appearing in the result set that have support below the threshold, can be ruled out just by means of a second pass over the data. In this way it would be possible to compute the actual support of the pairs to be output. This approach is always feasible when multiple passes over the data are possible, in this way turning false positives in a false problem.

For general similarity measures, like the ones described in Table 2.1, a smarter sampling technique will be necessary. The algorithm will take time $O(mb + \tau \sum_{0<i<j\leq n} s(i,j))$, where $s(i,j)$ is the similarity measure applied to the pair $\{i,j\}$, mb is the input size and τ is an input parameter. τ can be thought of as a constant that is used to tune the error probability of the algorithm; that is, the number of false positives and negatives. In particular τ plays a role in determining the number of samples taken for each pair. Looking at the time complexity we can see that the running time is either dominated by reading the input or is proportional to the sum of pairwise similarities, that is what we are trying to evaluate.

Chapter 2 digs out the details of what we described in this sub section.

1.3.2 Similarity mining - a streaming approach

We explain here what has been already described in an informal fashion in Section 1.1.2 with respect to sequential algorithms.

A variant of the problem described in the previous section considers the collection of subsets as a stream of transactions. According to what we said in Section 1.1.2 and what we will define more formally in Section 1.4.2, in the streaming model we cannot store the whole input and once we have dealt with one of the elements in the stream, that piece of the input is lost. In this setting, what we can store, is one transaction at a time. The implicit subset is the same of the former problem, namely $\binom{U}{2}$, since we will focus again on pairs. It is quite evident that in this setting, because of the fact that a second pass over the data is just impossible, producing candidates to be output is not an option and the algorithm must go directly for the interesting pairs.

The algorithm will exploit the same sampling technique and idea of the two passes version, adapted to deal with the constraints that streaming imposes.

Another question that arises quite naturally in this setting is how much space an algorithm must use in order to produce the desired output. We will answer to this question showing that no algorithm, even a randomized one, can do better than the trivial ones when trying to find the most similar pair in the input. For this reason, in order for the algorithm presented to be more efficient

than that, we will make the assumption that the transactions in the stream, arrive in random order.

The running time of the algorithm is a logarithmic factor away from linear, with high probability. In particular the time complexity has the form $O(mb \log(mn))$, where, again, mb is the input size, and m is the number of transactions, that is, the length of the stream. Within this time bound, the algorithm is able to spot pairs appearing frequently enough, a so called *support threshold*, and having similarity higher than a limit that depends either on n or on the input size. For what concerns the space usage, the dependency is either on the number n of distinct elements in the universe U, or on a parameter s. The algorithm, after sampling in a way that is similar to the one of Section 1.3.1, re-samples the pairs using a data structure that has size s, and influences the precision of the whole algorithm.

Chapter 4 brings the details of what we sketched in this sub section.

1.3.3 Frequent pairs in data streams

We extend here what has been already introduced in an informal fashion in Subsection 1.1.2 with respect to parallel algorithms.

The problem we address consists in reporting the most frequent pairs appearing in a stream of transactions. In particular we study this problem when the pairs frequencies follow a Zipfian distribution, defined as $f_i = C/i^z$ for the frequency f_i of the ith most frequent pair. Note that while we will consider parameter z to be a constant, this will not be the case for parameter C.

The problem of finding frequent items in an item stream has been deeply studied in many aspects, and several techniques have been developed depending on the specific function that has to be computed on the stream.

The set from which the implicit sampling is carried out is again the multiset of all pairs contained in all transactions. However, the main goal of the sampling in this case is achieving a small space usage by means of smart data structures. For what concerns the time complexity, the algorithm will actually produce the whole multiset; in order to avoid a quadratic cost, the algorithm will rely on the use of multiple processors, splitting the stream of pairs in sub-streams that are tackled in parallel.

To fulfil the requirement of small space usage and a good precision, the algorithm composes well known techniques for streams of items, extending them in order to be able to deal with pairs of items; moreover, as pointed out at the beginning of the section, the algorithm assumes that the frequencies of pairs follow a Zipfian distribution. The algorithm provides, for a space usage k, a Zipfian distribution $f_i = C/z^i$ and d distinct pairs in the stream, a constant error guarantee when reporting pairs whose frequency is above a threshold that depends on the parameters introduced before. The error then can be made arbitrarily small by means of Chernoff bounds, at the cost of running multiple instances of the algorithm.

Chapter 5 brings the details of what we introduced in this sub section.

1.3.4 Frequent traces of paths in graphs

A more formal definition of the graph mining problem described in Section 1.1.3 is: Given a directed acyclic graph $G = (V, E)$ and a labelling of the nodes in

V, find those sequences of node labels, called *traces*, of length at most m, so sequences of labels associated with paths of nodes of length at most m, such that they appear frequently in the graph. A question is: What is in this case the set from which we want to extract the frequent traces? This is the multiset S_m of all traces of length at most m existing in the graph, so the set of all subpaths of length at most m existing in the graph. The cardinality of this set is exponential in the maximum length m of traces that we allow. The algorithm we will present relies on the observation that traces have an inner recursive structure that can be exploited. As a matter of fact, let $S_i(v)$ denote the multiset of traces corresponding to paths (of length at most i) starting in a node $v \in V$. Then we can write:

$$S_i(v) = \{\text{label}(v)\} \times (\epsilon \cup \bigcup_{v'|(v,v')\in E} S_{i-1}(v'))$$

where ϵ is the empty trace. So, if m is the maximum length of a trace we are interested in, we can write:

$$S_m = \bigcup_{v\in V} S_m(v).$$

From S_m we will sample traces recursively and in a way that a branch of recursion will produce at least one sample. This fact will help us in bounding the running time used by the sampling, binding it to the number of samples taken. It is worth remarking that the running time will be independent of the total size of S_m.

Chapter 6 traces the details of what we depicted in this sub section.

1.3.5 Structure prediction

We explain here what has been already described in an informal fashion in Section 1.1.4.

The matrix multiplication problem we address consists in estimating the number of non zero entries in the result of the product of two boolean matrices. This setting captures the nature of various problems in databases and computer algebra. As a matter of fact, it is fairly intuitive to understand the formulation of the problem basing the description on databases foundations. Matrix multiplication can be used for data mining purposes in order to compute the support similarity measure introduced in Section 1.3.1 in the way that follows. Consider a $0-1$ matrix A in which each row represents an item and each column a transaction. It will be $a_{i,p} = 1$ if and only if item i appears in transaction p. Then, computing $\tilde{A} = A \times A^T$, we get that for an item i, $\mathit{support}(i) = \tilde{a}_{i,i}$. Because $\tilde{A}$ is symmetric, $\mathit{support}(i,j) = \tilde{a}_{i,j} = \tilde{a}_{j,i}$ for any item $j \neq i$.

Also the problem that we address can be turned into a data mining tool, since it allows for estimating the space usage of a run of Agrawal and Srikant's *A-Priori* algorithm [6]. This algorithm, as a first step, produces and stores the set of all pairs that can be generated by a collection of transactions. This number of distinct pairs, hence the space usage of the algorithm, can be estimated running our algorithm on matrices built as described previously, when the product between A and A^T is a boolean product. Figure 1.1 gives an example of both the cases that we have just discussed. An interesting application of this prop-

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}, \quad A^T = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix},$$

$$\tilde{A} = A \times A^T = \begin{cases} \begin{pmatrix} 3 & 2 & 1 & 2 \\ 2 & 2 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 2 \end{pmatrix} & \text{over } \mathbb{N} \cup \{0\} \\ \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} & \text{over boolean} \end{cases}.$$

Figure 1.1: *The rows of A represent the items, while the columns represent the transactions. The elements of $\tilde{A}$ are the support of single elements on the diagonal, and of pairs in other positions, when the product is computed over the natural numbers union* 0*. On the other hand, when the product is boolean, besides the diagonal always having* 1 *in all positions, the result has* 1 *in positions corresponding to pairs generated by* A-Priori*.*

erty is computing the cardinality of the result of the natural join of two relations followed by a projection that eliminates the columns used to join. To represent this in a mathematical fashion, consider two tables $R_1 = (a, b)$ and $R_2 = (b, c)$ where a, b and c are sets of attributes. Using standard relational algebra operators, what we want to compute is the cardinality of $R = \pi_{a,c}(R_1 \bowtie R_2)$, that is, the projection of the attributes a and c of the relation R obtained from the natural join of R_1 and R_2. It is immediately clear that the difficulty lies in the fact that the result is a set, so duplicates of the same row collapse to one and only one row. Notice that there can be as many as $|b|$ copies of a row once b gets projected out, where $|b|$ stands for the number of possible values that the set of attributes b can acquire.

In essence here the multiset from which we will sample, is the table $R_1 \bowtie R_2$, that is, the table having all rows before the projection takes place. This table can be very large, as of $|b| \cdot |R_1| \cdot |R_2|$ so, again, actually computing it is not a feasible task.

The algorithm we will present computes an approximation of the solution with small error probability, even $o(1)$ for any fixed error, using time that is, in expectation, *linear* in the size of the input, that is the sum of the number of rows of the two input relations.

Chapter 7 multiplies the details of what we introduced in this sub section.

1.4 Models and tools

We introduce and describe the tools that we use in our algorithms, providing here some general and formal definitions. Besides presenting some of the con-

cepts that we will use extensively throughout the whole thesis, we provide an overview of the scientific fields where these tools fall in. The reader who is interested in deepening his knowledge of the topics that we touch on in the remainder of the chapter, will find useful references inside the sections.

1.4.1 Randomized algorithms

In the proceeding we often referred to the need of performing a sampling of the implicit sets. This is necessary since the size of those sets is simply too large to actually deal with. In this subsection, we will introduce and discuss some characteristics of randomized algorithms in general. For a deeper description and coverage of the topic, the book of Motwani and Raghavan [82] is a precious resource.

Randomized algorithms, besides having some remarkable practical applications such as hash tables and signatures, play an important role in theoretical computer science. Not only they allow fast algorithms that guarantee accurate output, they have also posed some of the most interesting questions in the field; for example, we can think of the inclusion relationships between randomised and deterministic complexity classes. On the other hand, randomisation often hides the structure of a problem, relying instead on the properties of some probability distribution; in these cases, even though it provides efficiency, it does not open to any real understanding of the characteristics of the problem at hand.

Randomised algorithms can be classified into two main categories:

Monte Carlo: algorithms in this family *make errors*, which means that the answer provided by the algorithm may be the wrong one; these algorithms can present errors of two different kinds:

- **One sided error:** this kind of algorithm can produce errors only in one way; for decision problems, this translates in the fact that if the algorithm answers `Yes` (answers `No`) then the answer is correct for sure; on the other hand, if it answers `No` (respectively `Yes`) the answer can be wrong with some probability;
- **Two sided error:** this kind of algorithms has not the property of the former class. They can answer with the wrong output in all cases so that every output has a probability larger than zero of being wrong;

Las Vegas: the other side of coin are the Las Vegas algorithms. The algorithms in this class are always right; their drawbacks are the fact that the running time is a random variable and the fact that the algorithms in this class may not, in principle, terminate; usually Las Vegas algorithms are defined in a way that imposes them to admit an upper bound on the expected running time.

The algorithms that will be presented in this thesis are all Monte Carlo ones, in a broader sense than the former definition admits. The problems we address are not decision ones, so the definition above has to be adapted. Our results usually have the form of computing an approximation of the exact solution with some guarantee, that is, usually, being a $1 \pm \varepsilon$ factor away from correctness. The guarantee on the approximation holds with a small error probability, so that the algorithms can output a solution whose quality is worse than the one

guaranteed; the algorithms we will present are Monte Carlo in this sense, and can make errors that are both one sided and two sided.

In the discussion that follows we will reason about computational complexity classes. We assume one of the standard models of computation and skate over the details and we will not define precisely what a complexity class is at all. We assume the reader is familiar with these concepts. Also it would be very space consuming to detail precisely the tools necessary for more accurate considerations about computational complexity. The reader who wants to go deeper in this topic can find plenty of discussions in some of the most interesting books that have been published during the short life of computer science. We signal here just a selection of these excellent books: [12, 13, 49, 56, 63, 85].

Polynomial time Las Vegas problems define the class ZPP = RP ∩ Co-RP, where RP is the class of decision problems that admit a polynomial time Monte Carlo algorithm with one sided error on acceptance and error probability smaller than or equal to 1/2. So Co-RP contains those problem admitting the inverse error behaviour with respect to those in RP, that are algorithms that can error on rejection with error probability smaller than 1/2. From the former considerations it is clear how to come up with a Las Vegas algorithm for a problem, given RP and Co-RP: Just run two algorithms, one from each class, for the problem and as soon as they agree on an answer, that answer will be correct for sure. This also gets it across of why the running time of the obtained Las Vegas algorithm is a random variable and does not admit an upper bound for the worst case: The two algorithms may run for an infinite number of times without agreeing on the result.

In order to have a better idea of the relationships between complexity classes, we introduce the class BPP. This class is defined as including those decision problems that admit polynomial time randomized algorithms that output the correct answer with probability at least 3/4.

It is quite evident that ZPP ⊆ RP ⊆ BPP. Both the classes RP and BPP are meaningful if and only if there exist a perfect source of randomness that the algorithms can exploit. This question is actually relevant and refers to the existence of suitable and proper pseudorandom generators. Without entering the theory and the results concerning this topic, it is worth noticing that this entities are intensively studied and are object of rather vivid discussions in the theoretical computer science community (see for example [55, 57, 71]). Figure 1.2 gives a visual idea of how some of the complexity classes relate to each other. This whole hierarchy of complexity classes is actually very intriguing and poses some of the most interesting problems in current theoretical computer science:

- P = BPP?
- BPP ⊆ NP?
- RP = Co-RP?
- RP ⊆ NP ∩ Co-NP? (This would be implied by proving the former);

The first point has been the topic of many research papers during the years, and solving it would be a huge step forward in science.

In order to classify our algorithms we have to refer to optimisation problems rather than decision ones. In this framework, our algorithms all present a

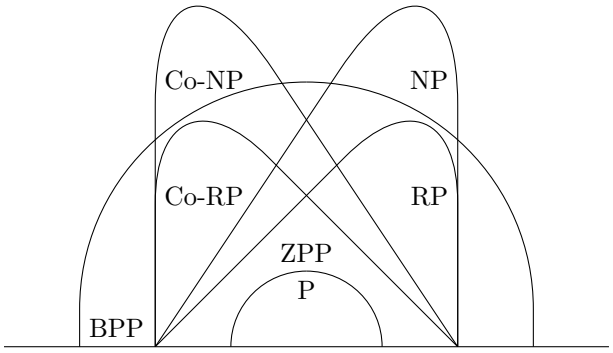


Figure 1.2: *The structure of some of the randomized and deterministic complexity classes. The inclusions are never strict, so, for instance, $P \subseteq BPP$.*

FPRAS (fully polynomial time randomized approximation scheme). As a matter of fact, the time bounds we will present are polynomial, even linear or quasi linear, in the input size and have a dependency on the approximation parameter ε that is polynomial in ε^{-1}.

Streaming problems have been a fruitful field for randomised algorithms. It is arguably infeasible to deal with streaming problems without using randomisation. Streaming has seen a huge increase in popularity in recent years. In fact, many papers on streaming have started to flow after the seminal work from Alon, Matias and Szegedi [7] has raised again the level of interest in the topic, after some years of calm. We will describe here the streaming framework in what follows. The description will be repeated later in the appropriate chapters when needed, in order to point out the salient elements of the model.

1.4.2 Streaming

In this sub section we give a formal definition of stream and characterise the streaming framework. For a wider, deeper coverage of the topic the reader can refer to [14, 83]. The streaming framework possesses the following characteristics:

- A stream is composed of data elements that arrive in an *on-line*[2] fashion;
- The order in which elements arrive is not under the control of the algorithm;
- The length of the stream is unknown and usually considered unbounded;
- Once an element of the data stream is processed, it is lost and cannot be accessed again, unless it has been *explicitly* recorded in memory.

[2]On-line is a setting where the input is not know *a priori*, and is discovered as elements arrive.

A formal definition of a stream can be given according to several models.

Definition 1.4.1. An input *stream* $a_1, a_2, \ldots$ arrives sequentially, one item at the time. The elements of the stream are such that $\forall i \,.\, a_i \in \{1, \ldots, N\}$. This is a set of objects, which can be pairs and are not necessarily numbers. We can think of the elements of the universe $\{1, \ldots, N\}$ as a mapping of values in $\mathbb{R}$ The sequence of items describes an underlying signal $\mathbf{S}$. The signal is essentially mapped by the elements of the stream, so that $\mathbf{S}$ can be considered a function such that: $\mathbf{S} : \{1, \ldots, N\} \to \mathbb{R}$.

According to how the stream represents $\mathbf{S}$ we have the following definitions:

Time series model: $\forall i \,.\, a_i = \mathbf{S}[i]$; this means that each element a_i of the stream is the representation of the i^{th} element of the signal;

Cash register model: $\forall i \,.\, a_i = (j, I_i)$, $I_i \geq 0$, so that $\mathbf{S}_i[j] = \mathbf{S}_{i-1}[j] + I_i$; this means that $\mathbf{S}_i$ represents the state of the signal after the i^{th} element of the stream has been seen; this model is quite popular and often used in literature when addressing streaming problems;

Turnstile model: $\forall i \,.\, a_i = (j, U_i)$, so that $\mathbf{S}_i[j] = \mathbf{S}_{i-1}[j] + U_i$; the difference with the former model stands in the fact that, in this case, the state of the signal gets updated and not only incremented, since U_i can be positive, negative or zero; there is a variant to this model, called the *strict* turnstile model, where it must be true that $\mathbf{S}[j] \geq 0$. ◦

Our algorithms use both the time series (Chapters 4,7) and the cash register models (Chapters 2,4,6). The former will be used when we do not care about the frequency of elements but will focus only on single appearances, while the second will be used whenever statistics about frequencies are necessary. It is interesting to point out how our algorithms often generate a stream from the input. Such a generated stream, often falls in the cash register model. Some of our algorithms could be extended to use the turnstile model, essentially considering a *fully dynamic* scenario. For example, consider the problem presented in Section 1.3.2, augmenting it in order to take into account the deletion of a transaction. This would be captured and naturally represented by the turnstile model, just by using negative updates.

The performances of an algorithm that computes some function of a signal can be measured in several ways:

- Time used per element a_i;
- Space used by the algorithm at any give time t;
- Total time used to compute the function at any given time t.

In this framework, we usually aim for (*i*) a very small time per element, representing the fact that data arrive at a high pace, (*ii*) small space, representing the fact that the algorithm runs on limited hardware, (*iii*) and small total time, which is not a surprising desideratum.

In order to achieve a succinct space usage, it is necessary to use algorithmic techniques to represent the input without storing it explicitly; that is, maintain a synopsis of the input. In order to adhere to this requirement, various approaches can be used, also depending on the function to be computed on the signal:

Random samples: This technique is the most immediate one can think of; noticeably it allows for further synopses to be computed on top of the sample. A good example are the *frequent items algorithms*; in this algorithms sampling is often used in connection with a data structure that keeps track of the counts of a subset of elements from the stream; these algorithm are also called *counter-based*, for obvious reasons. We will use some of these techniques in the algorithms that we will present in what follows;

Sketching: This technique has been introduced by the aforementioned paper of Alon, Matias and Szegedy [7]; the technique consists of maintaining a summary of the data that is able to represent some qualities of the stream that the algorithm wants to compute. The summary that is maintained has not, usually, an intuitive structure; it looks instead fairly involved from a semantic perspective, yet simple and easy to maintain. This kind of technique is particularly suited when the goal is to compute moments of the stream or, equivalently, norms of a vector in the turnstile or cash register model. There are also remarkable examples of the use of sketching in a frequent items scenario;

Histograms: This technique aims to represent the distribution of values in the stream, keeping the space usage within reasonable boundaries. Histograms allow for an even finer sub-classification; we will not detail the various techniques but the interested reader can refer to the cited literature, in particular [14], for further explanations and accurate references:

- V-optimal histogram;
- Equi-Width histogram;
- End-biased histogram.

Wavelets: This technique represents the signal using a suitable basis of orthogonal vectors; distinct wavelets differ according to how the basis is chosen;

Sliding windows: This is a largely unexplored area, where the academic literature is sparse. One area where it has been used to some extent, is data mining, as discussed in Section 4.1. The idea behind this model is to consider only the more recent elements in the stream, since those are good candidates for computing the goal function; the reason why the younger elements are considered more interesting is that they may hold a higher influence if they will continue to appear in the part of the stream yet to come. This approach makes sense only in some specific scenarios, where the past part of the stream is less relevant than the recent part.

Our algorithms use both random samples and sketching techniques, by maintaining counting data structures for the former (Chapters 2, 4 and 6) and synopsis data structures for the latter (Chapters 5 and 7). The other techniques are reported for the sake of completeness and to give the reader an overview of the field.

Another interesting aspect of streaming scenarios, arguably the most interesting aspect, is that they allow for many techniques to be used in order to

compute lower bounds. These techniques fall, in essence in three main categories:

Compressibility: Given a problem Q and a prefix of the stream P, suppose that such prefix is stored, in some compressed way, in a data structure A; we could then craft a suffix S of the stream such that, solving Q on $P \circ S$ could be used to reconstruct exactly P from A. Since the space used by A can compress only a subset of prefixes, the lower bound on the size of A follows; we use this technique in Chapter 5 to give a proof of infeasibility;

Communication complexity: This is a rather natural way of finding lower bounds for streaming algorithms. Usually one can reduce a two party protocol to the streaming problem, and since the protocol needs a certain amount of communication, the lower bound follows; we use his technique in Chapter 4 to get an unavoidable space usage need for the problem of mining similar pairs in a stream of transactions;

Reduction: This is the classic and well known technique, used in several distinct fields of theoretical computer science.

Now we will start the description of the techniques we developed in order to carry out sampling on implicit sets. All the algorithms presented are randomized Monte Carlo algorithms, using streaming tools or being themselves streaming algorithms. The main innovation of all the algorithms, stands in the way the sampling is carried out on the implicit set, and in the way we exploit the structure of the set itself.

1.5 Publishing information

Most of the material contained in the thesis has been published in or is currently submitted for publication to journals, international conferences and workshops. With respect to the published or submitted versions, some additions have been made, in order to clarify or extend the content of the papers.

On the other hand, the algorithmic technique presented in Chapter 3, is completely original and is published for the first time in this thesis. This technique abstracts several specialised methods that are used in Chapters 2, 4, 5 and 7.

In the following list, we give detailed publishing information in connection with each chapter:

Chapter 2: Andrea Campagna and Rasmus Pagh. Finding associations and computing similarity via biased pair sampling. *Knowledge and Information Systems* [22]. A shorter version of this paper can also be found in *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09)* [23];

Chapter 4: Andrea Campagna and Rasmus Pagh. On Finding Similar Items in a Stream of Transactions. *Proceedings of the 10th IEEE International Conference on Data Mining Workshops (ICDMW '10)* [25];

Chapter 5: Andrea Campagna, Konstantin Kutzkov and Rasmus Pagh. Frequent Pairs in Data Streams: Exploiting Parallelism and Skew. Submitted for publication;

Chapter 6: Andrea Campagna and Rasmus Pagh. On Finding Frequent Patterns in Event Sequences. *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM '10)* [24];

Chapter 7: Rasmus R. Amossen, Andrea Campagna and Rasmus Pagh. Better Size Estimation for Sparse Matrix Products. *Proceedings of the 13th International Workshop APPROX '10, and 14th International Workshop RANDOM '10* [8].

1.6 Acknowledgements

We wish to thank Blue Martini Software for contributing the KDD Cup 2000 data that we used for experiments in Chapter 2. It is opportune to point out that the work presented in Chapter 6 was funded in part by the SPOPOS project, supported by the Research and Innovation Agency under the Danish Ministry for Knowledge, Technology and Development. We wish to thank Eivind Abusland and Matyas Markovics for contributing to the Wikipedia data that we used in Chapter 5. We would like to thank Jelani Nelson for useful discussions, and in particular for introducing us to the idea of buffering to achieve faster data stream algorithms used in Chapter 7. In the same chapter, Rolf Fagerberg and Konstantin Kutzkov pointed out mistakes that have been corrected in the presented version, so we wish to thank them. Also, we thank Sumit Ganguly for clarifying the lower bound proof of [47] to us. A thank goes to David Raymond Christiansen and Phạm Dăng Ninh for suggestions improving the presentation of this thesis. Finally we wish to thank all the anonymous reviewers of our papers for pointing out mistakes and related work.

Chapter 2

BiSam - a Two Passes Approach

Sampling-based methods have previously been proposed for the problem of finding interesting associations in data, even for low-support items. While these methods do not guarantee precise results, they can be vastly more efficient than approaches that rely on exact counting. However, for many similarity measures no such methods have been known. In this chapter we show how a wide variety of measures can be supported by a simple *biased* sampling method. The method also extends to find high-confidence association rules. We demonstrate theoretically that our method is superior to exact methods when the threshold for "interesting similarity/confidence" is above the average pairwise similarity/confidence, and the average support is not too low. Our method is particularly advantageous when transactions contain many items. We confirm in experiments on standard association mining benchmarks that we obtain a significant speedup on real datasets. Reductions in computation time of over an order of magnitude, and significant savings in space, are observed.

2.1 Introduction

A central task in data mining is finding associations in a binary relation. Typically, this is phrased in a "market basket" setup, where there is a sequence of baskets (from now on "transactions"), each of which is a set of items. The goal is to find patterns such as "customers who buy diapers are more likely to also buy beer". There is no canonical way of defining whether an association is interesting — indeed, this seems to depend on problem-specific factors not captured by the abstract formulation. As a result, a number of measures exist: In this chapter we deal with some of the most common measures, including *Jaccard* [34], *lift* [19, 4], *cosine*, and *all_confidence* [74, 84]. In addition, we are interested in high-confidence association rules, which are closely related to the *overlap coefficient* similarity measure. We refer to [59, Chapter 5] for general background and discussion of similarity measures.

In the discussion we limit ourselves to the problem of binary associations, i.e., patterns involving pairs of items. There is a large literature considering the challenges of finding patterns involving larger itemsets, taking into account the

aspect of time, multiple-level rules, etc. Previous methods rely on one of the following approaches:

1. Identifying item pairs (i, j) that "occur frequently together" in the transactions — in particular, this means counting the number of co-occurrences of each such pair — or

2. Computing a "signature" for each item such that the similarity of every pair of items can be estimated by (partially) comparing the item signatures.

Our approach is different from both these approaches, and generally offers improved performance and/or flexibility. In some sense we go directly to the desired result, which is the set of pairs of items with similarity measure above some user-defined threshold Δ. Our method is *sampling* based, which means that the output may contain false positives, and there may be false negatives. However, these errors are rigorously understood, and can be reduced to any desired level, at some cost of efficiency — our experimental results are for a false negative probability of less than 2%. The sampling method that we describe is the main novelty of the approach. It samples, as it will be clear in the following, pairs from the multiset of all pairs that appear in all transactions.

The main focus in many previous association mining publications has been on *space usage* and the number of *passes* over the dataset, since these have been recognized as main bottlenecks. We believe that time has come to also carefully consider *CPU time*. A transaction with b items contains $\binom{b}{2}$ item pairs, and if b is not small the effort of considering all pairs is non-negligible compared to the cost of reading the itemset. This is true in particular if data reside in RAM, or on a modern SSD that is able to deliver data at a rate of more than a gigabyte per second. One remedy that has been used (to reduce space, but also time) is to require *high support*, i.e., define "occur frequently together" such that most items can be thrown away initially, simply because they do not occur frequently enough (they are below the *support threshold*). However, as observed in [34] this means that potentially interesting or useful associations (e.g. correlations between genes and rare diseases) are not reported. We consider instead the problem of finding associations *without* support pruning. Of course, support pruning can still be used to reduce the size of the dataset before our algorithms are applied.

In the following sections we first discuss the need for focusing on CPU time in data mining, and then elaborate on the relationship between our contribution and related works.

2.1.1 I/O versus CPU

In recent years, the capacity of very fast storage devices has exploded. A typical desktop computer has 4–16 GB of RAM, that can be read (sequentially) at a speed of at least 800 million 32-bit words per second. The flash-based ioDrive Duo of Fusion-io offers up to over a terabyte of storage that can be read at around 400 million 32-bit words per second. Thus, even massive datasets can be read at speeds that make it challenging for CPUs to keep up. An 8-core system must, for example, process 100 million (or 50 million) items per core per second. At 3 GHz this is 33 clock cycles (or 66 clock cycles) per item. This means

that any kind of processing that is not constant time per item (e.g., using time proportional to the size of the transaction containing the item) is likely to be CPU bound rather than I/O bound. For example, a hash table lookup requires on the order of 5-10 ns even if the hash table is L2 cache-resident (today less than 10 MB per core). This gives an upper limit of 100-200 million lookups per second in each core, meaning that any algorithm that does more than a dozen hash table operations per item (e.g. updating the count of some item pairs) is definitely CPU bound, rather than I/O bound. In conclusion, we believe it is time to carefully consider optimizing internal computation time, rather than considering all computation as "free" by only counting I/Os or number of passes. Once CPU efficient algorithms are known, it is likely that the remaining bottleneck is I/O. Thus, we also consider I/O efficient versions of our algorithm.

2.1.2 Previous work

Exact counting of frequent itemsets

The approach pioneered by the A-Priori algorithm [5, 6], and refined by many others (see e.g. [54, 78, 20, 86, 88]), allows, as a special case, finding all item pairs (i, j) that occur in more than k transactions, for a specified threshold k. However, for the similarity measures we consider, the value of k must in general be chosen as a low constant, since even pairs of very infrequent items can have high similarity. This means that such methods degenerate to simply counting the number of occurrences of all pairs, spending time $\Theta(b^2)$ on a transaction with b items. Also, generally the space usage of such methods (at least those requiring a constant number of passes over the data) is at least 1 bit of space for each pair that occurs in *some* transaction. An experimental comparison for some 2004 state-of-the-art algorithms for frequent itemset mining is carried out in [53].

The problem of counting the number of co-occurrences of all item pairs is in fact equivalent to the problem of multiplying sparse 0-1 matrices. This equivalence has already been showed in Section 1.3.5, but we present it again, for ease of understanding: consider the $n \times m$ matrix A in which each row A_i is the incidence vector having 1 in position p iff the ith element in the set of items appears in the pth transaction. Each entry $\tilde{\mathrm{A}}_{i,j}$ of the $n \times n$ matrix $\tilde{\mathrm{A}} = \mathrm{A} \times \mathrm{A}^T$ represents the number of transactions in which the pair (i, j) appears. The best theoretical algorithms for (sparse) matrix multiplication [9, 36, 96] scale better than the A-Priori family of methods as the transaction size gets larger, but because of huge constant factors this is so far only of theoretical interest.

Sampling methods

Toivonen [90] investigated the use of sampling to find candidate frequent pairs (i, j): Take a small, random subset of the transactions and see what pairs are frequent in the subset. This can considerably reduce the memory used to actually count the number of occurrences (in the full set), at the cost of some probability of missing a frequent pair. This approach is good for high-support items, but low-support associations are likely to be missed, since few transactions contain the relevant items.

Cohen and Lewis [35] present an algorithm for approximate matrix multiplication that can be used for finding similar pairs (in the same approximate sense

that we pursue here) according to some similarity measures such as *cosine* and *lift*. In fact, for these similarity measures their algorithm will produce estimator random variables with the same distribution as ours (binomial), in a similar time bound. Our approach handles more general similarity measures, uses less space in addition to the input, and we couple the sampling with a space-efficient algorithm for finding the most similar pairs.

Locality-sensitive hashing

Cohen et al. [34] proposed the use of another sampling technique, called *min-wise independent hashing*, where a small number of occurrences of each item (a "signature") is sampled. This means that occurrences of items with low support are more likely to be sampled. As a result, pairs of (possibly low-support) items with high *jaccard coefficient* are found — with a probability of false positives and negatives. A main result of [34] is that the time complexity of their algorithm is proportional to the sum of all pairwise jaccard coefficients, plus the cost of initially reading the data. Our main result exactly the same form, but has the advantage of supporting a wide class of similarity measures.

Min-wise independent hashing belongs to the class of *locality-sensitive* hashing methods [65]. Another such method was described by Charikar [29], who showed how to compute succinct signatures whose Hamming distance reflects angles between incidence vectors. This leads to an algorithm for finding item pairs with cosine similarity above a given threshold (again, with a probability of false positives and negatives), that uses linear time to compute the signatures, and $\Theta(n^2)$ time to find the similar pairs, where n is the number of distinct items in all transactions. Charikar also shows that many similarity measures, including some measures supported by our algorithm, cannot be handled using the approach of locality-sensitive hashing.

Deterministic signature methods

In the database community, finding all pairs with similarity above a given threshold is sometimes referred to as a "similarity join." Recent results on similarity joins include [11, 30, 93, 94]. While not always described in this way, these methods can be seen as deterministic analogues of the locality-sensitive hashing methods, offering *exact* results. The idea is to avoid computing the similarity of every pair by employing succinct "signatures" that may serve as witnesses for low similarity. Most of these methods require the signatures of every pair of items to be (partially) compared, which takes $\Omega(n^2)$ time. However, the worst-case asymptotic performance appears to be no better than the A-Priori family of methods. A similarity join algorithm that runs faster than $\Omega(n^2)$ in some cases is described in [11]. However, this algorithm exhibits a polynomial dependence on the maximum number k of differences between two incidence vectors that are considered similar, and for many similarity measures the relevant value of k may be linear in the number m of transactions.

Larger significant itemsets

Wu et al. [92] consider mining of significant itemsets according to a measure related to *lift*. In particular, their approach extends to negative associations.

Zhang et al. [98] use the same approach, presenting also a fuzzy variant. However, both approaches require exact counting of the number of co-occurrences of all itempairs (where each item is above the support threshold). Therefore the performance for finding significant/similar *pairs* is similar to the performance of other exact counting methods.

Streaming algorithms

In Chapter 4, a modification of the technique presented in this chapter finds similar pairs in a randomly ordered stream of transactions. In the streaming framework one can see and store only one transaction at a time. In order to tackle the difficulties of the new environment in an efficient fashion, that is avoiding to sample too many pairs, a support threshold is used. Moreover the sampling is carried out in a slightly different way, in order to guarantee that it happens in "real time". Also, we replace the final step of the algorithm by the method of [41] that is particularly well suited to the setting of randomly ordered data. Chapter 4 also contains a lower bound on the space that every algorithm mining similar k-itemsets must use, in a worst case scenario, extending a previous lower bound in [40] and explaining why the assumption concerning the transactions arriving in random order is necessary.

2.1.3 Our results

As noted previously in the chapter, we present a novel sampling technique to handle a variety of measures (including jaccard, lift, cosine, and all_confidence), even finding similar pairs among low support items. The idea is to sample a subset of all pairs (i, j) occurring in the transactions, letting the sampling probability be a function of the supports of i and j, such that the expected number of times a pair is sampled is proportional to $s(i, j)$. Given a threshold Δ the sampling rate can be scaled such that any pair with similarity above Δ is likely to be sampled several times, whereas pairs with similarity "far below" Δ are likely not to be sampled. The number of times a pair is sampled follows a binomial distribution, which allows us to use the sample, in a *filtering* phase, to infer which pairs are likely to have similarity above the threshold, with rigorous bounds on false negative and false positive probabilities.

A naïve implementation of this idea would still use quadratic time for each transaction, but we show how to do the sampling time that is *linear* in the size of the transaction and number of sampled pairs. In turn, the expected number of samples is proportional the sum of all pairwise similarities between items. We will argue that this running time is the best one could hope for with no conditions on the distribution of pairwise similarities. Under reasonable assumptions, e.g. that the average support is not too low, this gives a speedup of a factor $\Omega(b)$, where b is the average size of a transaction, compared to exact counting methods.

We show in extensive experiments on standard datasets for testing data mining algorithms that our approach (with sampling rate resulting in a 1.8% false negative probability) gives speedup factors in the vicinity of an order of magnitude, as well as significant savings in the amount of space required, compared to exact counting methods. We also present evidence that for datasets with many

distinct items, our algorithm may perform significantly less work than methods based on locality-sensitive hashing 2.4.1.

2.1.4 Notation and framework

Let $T_1, \ldots, T_m$ be a sequence of transactions, $T_j \subseteq [n] = \{1, \ldots, n\}$. For $i = 1, \ldots, n$ let $S_i = \{j \mid i \in T_j\}$, i.e., S_i is the set of occurrences of item i.

We are interested in finding associations among items, and consider a framework that captures the most common measures from the data mining literature. Specifically, we can handle a similarity measure $s(i, j)$ if there exists a function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{R}^+$ that is non-increasing in both parameters, and such that:

$$s(i, j) = |S_i \cap S_j| \, f(|S_i|, |S_j|) \ .$$

Table 2.1 shows particular measures that fall within this framework. The monotonicity requirement on f holds for any reasonable similarity measure: It says that for a given value of $|S_i \cap S_j|$, adding an occurrence of i or j should not increase the similarity. In the following we assume that f is computable in constant time, which is clearly a reasonable assumption for the measures of Table 2.1. In the time analysis we will further assume that f is *polynomial* in the sense that changing the input by a constant changes the value of f by a constant, specifically that $f(c_1, c_2) = O(f(c_1/2, c_2/2))$ for all c_1, c_2. This clearly holds for all similarity measures we consider, and arguably for any reasonable similarity measure.

We end with some observations on other measures that can be handled directly or indirectly by our framework.

Composite measures

Notice that if $f_1(|S_i|, |S_j|)$ and $f_2(|S_i|, |S_j|)$ are both non-increasing, then any linear combination $\alpha f_1 + \beta f_2$, where $\alpha, \beta > 0$, is also non-increasing. Similarly, $\min(\alpha f_1, \beta f_2)$ is non-increasing. This allows us to use BiSam to directly search for pairs with high similarity according to several measures (corresponding to f_1 and f_2), e.g., pairs with cosine similarity at least 0.7 *and* lift at least 2.

Handling the Jaccard measure

We observe the following relationship between the jaccard and dice similarity measures: $s_{\text{jaccard}}(i, j) = s_{\text{dice}}(i, j)/(1 - s_{\text{dice}}(i, j))$. Observe that s_{jaccard} grows monotonically with s_{dice}, and that the derivative wrt. s_{dice} is in the range $[1; 4]$. This means that most questions about jaccard similarity can be translated into questions about dice similarity. For example, if we are interested in all pairs with a certain jaccard similarity, this translates directly into all pairs with a certain dice similarity.

2.2 The BiSam algorithm

For a given parameter $\tau > 0$ our goal is to sample pairs of items such that (i, j) is sampled $\tau \cdot s(i, j)$ times in expectation. Also, we want the occurrences of an item pair to be sampled independently, such that the number of samples follow a (highly concentrated) binomial distribution.

Measure	$s(i,j)$	$f(\lvert S_i\rvert,\lvert S_j\rvert)$
lift	$m\frac{\lvert S_i\cap S_j\rvert}{\lvert S_i\rvert\lvert S_j\rvert}$	$m/(\lvert S_i\rvert\cdot\lvert S_j\rvert)$
cosine	$\frac{\lvert S_i\cap S_j\rvert}{\sqrt{\lvert S_i\rvert\lvert S_j\rvert}}$	$1/\sqrt{\lvert S_i\rvert\cdot\lvert S_j\rvert}$
all_confidence	$\frac{\lvert S_i\cap S_j\rvert}{\max(\lvert S_i\rvert,\lvert S_j\rvert)}$	$1/\max(\lvert S_i\rvert,\lvert S_j\rvert)$
dice	$\frac{\lvert S_i\cap S_j\rvert}{\lvert S_i\rvert+\lvert S_j\rvert}$	$1/(\lvert S_i\rvert+\lvert S_j\rvert)$
overlap_coef	$\frac{\lvert S_i\cap S_j\rvert}{\min(\lvert S_i\rvert,\lvert S_j\rvert)}$	$1/\min(\lvert S_i\rvert,\lvert S_j\rvert)$

Table 2.1: *Some measures covered by our algorithm and the corresponding functions. Note that finding all pairs with overlap coefficient at least Δ implies finding all association rules with confidence at least Δ, where confidence$(i,j) = \frac{\lvert S_i\cap S_j\rvert}{\lvert S_i\rvert}$ and is not symmetric. The* Jaccard *similarity measure can be handled via the dice measure, as argued in Section 2.1.4.*

The output of our algorithm, named BiSam (for *bi*ased *sam*pling), will be an unordered sequence of samples. It will be convenient to work with *weighted* samples, i.e., with each sampled pair we associate a positive real number (which will be at least 1, but not necessarily integer). We define the number of occurrences of a pair (i,j) as the sum of the associated numbers.

We observe that all measures in Table 2.1 are symmetric, $s(i,j) = s(j,i)$, so it suffices to sample either (i,j) or (j,i). Our pseudocode will make this optimization, by dealing with *sets* $\{i,j\}$, but can easily be modified to also handle asymmetric measures.

2.2.1 Algorithm idea

Algorithm 2.1 shows pseudocode for BiSam. In an initial pass over the data it computes item support counts, stored in a (hash) map c. For convenience, we precompute item counts rounded down to the nearest power of 2, stored as c'.

After the initial pass, the algorithm iterates through the transactions once more. For each transaction T_t, some number of size-2 subsets of T_t are output, with a weight associated with each pair. The processing of a transaction starts with sorting the items according to value of c', i.e., they are "roughly sorted" according to support. Below, we discuss how this can be done in linear time by exploiting that c' has only $\lceil \log m \rceil$ possible distinct values.

The main loop of the algorithm, Lines 4-16, outputs those pairs $\{T_t[i], T_t[j]\}$ for which $f(c(T_t[i]), c(T_t[j]))\tau > r$, where r is a random real number in $[0;1)$. This can be seen as follows. For each value of i the algorithm iterates through $j = i+1, i+2, \ldots$ until $j = |T_t|$ or $f(c'(T_t[i]), c'(T_t[j]))\tau \leq r$. In both cases we can conclude, since f is non-increasing, that no more pairs with the current value of i should be reported. The total time for the outer loop, Line 7, is $O(|T_t|)$, and the time for the inner loop, Lines 9-14, is proportional to the number of pairs $\{i,j\}$ for which $f(c'(T_t[i]), c'(T_t[j]))\tau < r$.

A pair $\{T_t[i], T_t[j]\}$ is sampled with probability $\min(1, f(c(T_t[i]), c(T_t[j]))\tau)$. In cases where we sample with probability less than 1, we associate a weight of 1 with the sample; otherwise we assign the weight $f(c(T_t[i]), c(T_t[j])\tau$. In either

case, the expected weight assigned to the sample is $f(c(T_t[i]), c(T_t[j])\tau$. Thus we have the following:

Lemma 2.2.1. *Let $M(i,j)$ denote the total weight of the pair $\{i,j\}$ in the output of* BiSam. *For all pairs (i,j), where $i \neq j$ and $c(i) \leq c(j)$, if $f(c(i), c(j))\tau < 1$ then at the end of the procedure, $M(i,j)$ has binomial distribution with $|S_i \cap S_j|$ trials and mean*

$$|S_i \cap S_j| f(|S_i|, |S_j|)\tau = s(i,j)\tau.$$

If $f(c(i), c(j))\tau \geq 1$ then at the end of the procedure $M(i,j) = s(i,j)\tau$ with probability 1. ∘

In other words, the weight $M(i,j)$ is an unbiased estimator for $s(i,j)\tau$ that is tightly concentrated. This allows us to give firm guarantees on the probability that $M(i,j)/\tau$ deviates substantially from $s(i,j)$, as discussed in Section 2.3.1.

```
 1: procedure BiSam(T_1, ..., T_m; f, τ)
 2:     c := ItemCount(T_1, ..., T_m)
 3:     c' := FloorToPowerOf2(c)
 4:     for t := 1 to m do
 5:         sort T_t[] s.t. c'(T_t[j]) ≤ c'(T_t[j+1]) for 1 ≤ j < |T_t|
 6:         let r be a random number in [0;1)
 7:         for i := 1 to |T_t| do
 8:             j := i + 1
 9:             while j ≤ |T_t| and f(c'(T_t[i]), c'(T_t[j]))τ > r do
10:                 if f(c(T_t[i]), c(T_t[j]))τ > r then
11:                     output {{T_t[i], T_t[j]}, max(1, f(c(T_t[i]), c(T_t[j]))τ)}
12:                 end if
13:                 j := j + 1
14:             end while
15:         end for
16:     end for
17: end procedure
```

Algorithm 2.1: *Pseudocode for the* BiSam *algorithm. The call to the procedure* ItemCount$(\cdot)$ *on Line 2, returns a function (hash map) that contains the number of occurrences of each item. The call to the procedure* FloorToPowerOf2(c)*, on Line 3, returns a function that is obtained from c by rounding down occurrence counts to the nearest integer power of 2. $T_t[j]$ denotes the jth item in transaction number t.*

2.2.2 Implementation

The best implementation of the subprocedure ItemCount depends on the relationship between available memory and the number n of distinct items. If there is sufficient internal memory, it can be efficiently implemented using a hash table. In the following we first consider the standard model (often referred to as the "RAM model"), where the hash tables fit in internal memory, and assume that each insertion takes constant time. Then we consider the I/O

item	occurences	item	occurrences
i	$c(i)$	i	$c(i)$
1	66	6	31
2	66	7	28
3	65	8	5
4	60	9	5
5	58	10	3

Table 2.2: *Items in the example, with corresponding* ItemCount *values.*

item	FloorToPowerOf2	bucket	item	FloorToPowerOf2	bucket
i	$c'(i)$		i	$c'(i)$	
1	64	5	6	16	3
2	64	5	7	16	3
3	64	5	8	4	2
4	32	4	9	4	2
5	32	4	10	2	1

Table 2.3: *The table represents the function c' obtained by the application of the functional FloorToPowerOf2 to the function c. Hence the elements are grouped in 5 buckets.*

model, for which an I/O efficient "sort-and-count" implementation is discussed (Section 2.2.3).

The use of the standard implementation of the sorting step would require time $O(|T_t| \log |T_t|)$. However, we observe that there are only $\lceil \log m \rceil$ possible values of c', so this can be done more efficiently by bucket sorting (one bucket per value). In case $|T_t| < \log m$ we need a few extra tricks to get a linear time algorithm. We stress that these tricks are described for the purpose of the theoretical result, and are unlikely to yield an advantage in practice due to increased constant factors.

We modify standard bucket sort as follows: The buckets should be initialized in a lazy fashion, such that we do not use time on buckets that contain no elements. Also, when traversing the buckets to form the result we should not spend time on empty buckets. This can be achieved by maintaining a bit vector of length $\lceil \log m \rceil$ indicating which buckets are nonempty. Then the non-empty buckets can be found in time $O(|T_t|)$ using a constant-time least-significant-bit computation.

Example 2.2.1. Suppose ItemCount has been run and the supports of items 1–6 are as shown in Table 2.2. Table 2.3 shows the values associated to each element by the function c'.

Suppose now that the transaction $T_t = \{6, 4, 5, 3, 2, 1\}$ is given. Note that its items are written according to the mapping given by the function c'. Assuming the similarity measure is *cosine*, $\tau = 14$, and r for this transaction equal to 0.9, the algorithm would sample from $T_t \times T_t$ the pairs shown in Table 2.4. ◦

i	j	$f(c(i), c(j))\tau$	i	j	$f(c(i), c(j))\tau$
10	9	3.61	10	2	0.99
10	8	3.61	10	1	0.99
10	7	1.52	9	8	2.8
10	6	1.45	9	7	1.18
10	5	1.06	9	6	1.12
10	4	1.04	8	7	1.18
10	3	1.00	8	6	1.12

Table 2.4: *Pairs selected from T_t in the example. Notice that after realising the bucket pair $(2, 5)$ does not satisfy the inequality $f(c'(9), c'(3))\tau > r$, the algorithm will not take into account the pairs of bucket $(2, 4)$. Moreover, since $f(c'(7), c'(6))\tau < r$ the pairs of buckets $(3, 3), (3, 4), (3, 5), (4, 4), (4, 5)$ and $(5, 5)$.*

2.2.3 Analysis of running time

We provide a running time analysis both in the standard (RAM) model and in the I/O model of [3]. In the latter case we present an external memory efficient implementation of the algorithm, IOBiSam. Let b denote the average number of items in a transaction, i.e., there are bm items in total.

Running time in the standard model

The first part of the algorithm just goes through the input, using expected time $O(mb)$. The sorting of a transaction with b_1 items, performed as described above, takes $O(b_1)$ time, and in particular the total cost of all sorting steps is $O(mb)$. Similarly, the total cost of iterating through all transactions is $O(mb)$ if the cost of the **while** loop of Lines 9-14 is not counted.

The time for the **while** loop is proportional to the number of pairs $\{i, j\}$ for which $f(c'(T_t[i]), c'(T_t[j]))\tau < r$. That is, the probability that we spend time $O(1)$ on the pair $\{i, j\}$ is $\min(1, f(c'(T_t[i]), c'(T_t[j]))\tau)$. Summing over all pairs and all transactions we get an expected cost of at most:

$$\sum_t \sum_{\{i,j\}\subseteq T_t} f(c'(i), c'(j))\tau = O\left(\sum_t \sum_{\{i,j\}\subseteq T_t} f(c(i), c(j))\tau\right)$$

using the assumption that f is polynomial. Reordering the terms of the sum we get an expected cost of:

$$\sum_{\{i,j\}} |S_i \cap S_j| f(c(i), c(j))\tau = \sum_{\{i,j\}} s(i, j)\tau \ .$$

Theorem 2.2.1. *Suppose we are given transactions $T_1, \ldots, T_m$, each a subset of $[n]$, with mb items in total, and that f is a polynomial function such that $s(i, j) = |S_i \cap S_j| f(|S_i|, |S_j|)$. Then the expected time complexity of* BiSam$(T_1, \ldots, T_m; f, \tau)$ *in the standard model is $O\left(mb + \tau \sum_{1\le i<j\le n} s(i, j)\right)$.* ∘

Discussion. In most of our experiments the first of the two terms dominated the time complexity. This means that the running time is close to optimal, as $O(mb)$ is the time for just reading the input. However, we also found that for some datasets with mainly low-support items, the second term (the cost of reporting samples) dominated.

A comparison can be made with the complexity of schemes counting the occurrences of all pairs. Such methods use time $\Omega(mb^2)$, which is a factor $\Omega(b)$ larger than the first term. In fact, the difference will be larger if the distribution of transaction sizes is not even.

Similarity threshold. The parameter τ should be chosen such that $s(i,j)\tau$ is not too small, e.g. $s(i,j)\tau$, for the pairs that are considered highly similar. It is instructive to parametrise in terms of the threshold Δ for "interesting similarity". To ensure that interesting pairs are reported with good probability, τ must be chosen such that $\tau\Delta$ is not too small, e.g. in our experiments we use $\tau\Delta \approx 15$. The reason for this choice is explained in Section 2.4.1 and in Table 2.5.

A reasonable assumption is that Δ is greater than the average similarity, i.e., $\Delta \geq \sum_{1\leq i<j\leq n} s(i,j)/\binom{n}{2}$. In many cases Δ will be much greater than the average similarity, as discussed in Section 2.4.1. But just using the above we can obtain the following simple (in some cases pessimistic) upper bound on the time complexity:

Corollary 2.2.1. *If $\Delta = O(1/\tau)$ is no smaller than the average pairwise similarity, then expected time complexity of* BiSam *is $O(mb + n^2)$.* ∘

This means that under the assumption of the corollary we win a factor of at least $\min(b, m(b/n)^2)$ compared to the exact counting approach. If we let $\sigma = mb/n$ denote the average support, the speedup can be expressed as $\Omega(b\min(1, \sigma/n))$. So if the average support is n or more, we gain a factor $\Omega(b)$.

Independent items. As further evidence for (or explanation of) why the time complexity of the second term may be close to linear, we consider an input where each item i appears in a given transaction with probability p_i, independently of all other items. Thus, the probability that distinct items i and j appear in a transaction is p_ip_j. We observe that each similarity measure $s(i,j)$ in Table 2.1, with the exception of *lift*, satisfies $s(i,j) \leq \bar{s}(i,j)$, where $\bar{s}(i,j) = \frac{|S_i \cap S_j|}{|S_i|} + \frac{|S_i \cap S_j|}{|S_j|}$. Thus, we get an upper bound on running time for these measures by considering the similarity measure $\bar{s}(i,j)$. Observe that the expected value of $\bar{s}(i,j)$ is p_i+p_j by linearity of expectation. Hence, the expected sum of similarities is:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} p_i + p_j \leq \sum_{i=1}^{n} p_i n + \sum_{j=1}^{n} np_j = 2n \ .$$

This means that the running time of BiSam is $O(mb + \tau n)$ for independent items. Usually $mb \gg \tau n$, so the first term dominates.

Running time in the I/O model

We now present IOBiSam, an I/O efficient implementation of the BiSam algorithm. The rest of the chapter can be read independently of this section. As before, we assume the similarity measure is represented as $s(i,j) = |S_i \cap S_j| f(|S_i|, |S_j|)$.

In order to compute the support of each item, which means computing the ItemCount function, a sorting of the dataset's items is carried out. It is necessary to keep track of which transaction each item belongs to. To compute the sorted list of items, $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M})$ I/Os are needed [3], where $N = mb$ is the number of pairs $c = \langle$item, Transaction ID$\rangle$, M is the number of such pairs that fit in memory, and B is the number of pairs that fit in a memory page. When the items are sorted, it is trivial to compute the number of occurrences of each item, so it takes just $O(\frac{N}{B})$ I/Os to compute and store the tuples $c\langle$item,support,Transaction ID$\rangle$.

We then sort the tuples according to transaction ID, and secondarily according to support, again using $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M})$ I/Os. This gives us each transaction in sorted order, according to item supports. Assuming that each transaction fits in main memory[1] it is simple to determine which pairs satisfy the inequality $f(c(T_t[i]), c(T_t[j]))\tau > r$. When a pair satisfies the inequality, it is output, togheter with its weight $\max\{1, f(c(T_t[i]), c(T_t[j]))\tau\}$. This operations have a cost of $O(N/B)$ I/Os for the reads.

The most expensive steps are the sorting steps, implying that the following theorem holds:

Theorem 2.2.2. *Suppose we are given transactions $T_1, \ldots, T_m$, each a subset of $[n]$, with $N = mb$ items in total, and f is the function corresponding to the similarity measure s. Also let $|S_i \cap S_j| f(|S_i|, |S_j|) = s(i,j)$. The expected complexity of* IOBiSam$(T_1, \ldots, T_m; f, \tau)$ *in the I/O model is*

$$O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right) \text{ I/Os} \quad .$$

∘

2.3 How to use the BiSam output

Summing up the weight $M(i,j)$ of a given pair $\{i,j\}$ in the output of BiSam gives us, by Lemma 2.2.1, one of the following:

- Exactly $s(i,j)\tau$ with probability 1, or
- The value of a random variable with binomial distribution and expectation $s(i,j)\tau$.

In the former case we obviously know the similarity of i and j. In the latter case we can use statistical methods to derive bounds on likely and unlikely

[1] The assumption is made only for simplicity of exposition, since the result holds also without this assumption.

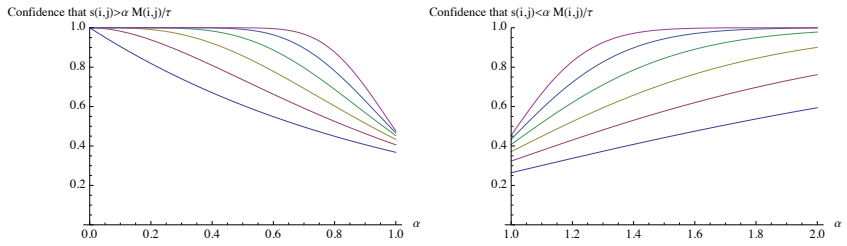


Figure 2.1: *Illustration of confidence levels, using Poisson approximation, for sample counts 1, 2, 4, 8, 16, and 32. Larger sample counts yield larger confidence.*

values of $s(i,j)$. In our theoretical discussion we will make use of the fact that the binomial distribution is closely approximated by the Poisson distribution (with the same mean) whenever the sampling probability is much smaller than the expectation. However, we stress that in concrete cases it is possible to do confidence calculations directly on the binomial distribution to get more accurate results.

Figure 2.1 shows confidence bounds for various observed values of a Poisson distributed random variable. We know that the mean value of $M(i,j)$ is $\tau s(i,j)$, so $M(i,j)/\tau$ is an unbiased estimator for $s(i,j)$. How likely is it that $s(i,j) < \alpha M(i,j)/\tau$ for some $\alpha < 1$? This depends on α and $M(i,j)$ — Figure 2.1 considers the cases where $M(i,j) \in \{1,2,4,8,16,32\}$. For each value of $M(i,j)$ the graph plotted is the probability of not observing a value as large as $M(i,j)$ given that $s(i,j) = \alpha M(i,j)/\tau$. This is the "confidence" we have in the assertion that $s(i,j) > \alpha M(i,j)/\tau$. Larger values of $M(i,j)$ yield higher confidences. Taking $M(i,j) = 8$ as an example we see that with 90% confidence the estimate $M(i,j)/\tau$ is at most $s(i,j)/0.59 \approx 1.7s(i,j)$, and with 90% confidence $M(i,j)/\tau$ is at least $s(i,j)/1.65 \approx 0.6s(i,j)$.

2.3.1 Errors with respect to a reporting threshold

One case we will consider in particular is when there is a threshold Δ such that we are interested in reporting all pairs with similarity Δ or more. To report such pairs with reasonable probability we cannot simply choose the pairs with weight $\Delta\tau$ or more, since this would give too many *false negatives*, i.e., pairs with $s(i,j) \geq \Delta$ that are not reported. The false negative probability can be decreased by lowering the weight threshold. In the following we assume that pairs with weight $\Delta\tau/2$ or more are reported.

Analysis of false negative probability. We first bound the probability that a pair $\{i,j\}$ with $s(i,j) \geq \Delta$ is not reported by the algorithm. This happens if $M(i,j) \leq \Delta\tau/2$ and $M(i,j)f(c(i),c(j),\Delta) < 1$. If $f(c(i),c(j))\tau \geq 1$ then the pair $\{i,j\}$ is reported with probability 1. Otherwise, since $M(i,j)$ has binomial distribution, it follows from Chernoff bounds (see e.g. [82, Theorem 4.2] with $\delta = 1/2$) that the probability of the former event is at most $\exp(-\delta^2\mu/2) = \exp(-\mu/8)$. Solving for μ this means that we have error probability at most ε if $\mu \geq 8\ln(1/\varepsilon)$. This bound is pessimistic, especially when ε is not very small. Tighter bounds can be obtained using the Poisson approximation to the

$\tau\Delta$	ε	ε'
3	0.199	0.0498
5	0.125	0.00674
10	0.0671	0.0000454
15	0.0180	$< 10^{-6}$
20	0.0108	$< 10^{-8}$
30	0.00195	$< 10^{-13}$

Table 2.5: *Values of $\tau\Delta$ and corresponding error probabilities ε. The error probabilities ε' are for the variant of the algorithm where we return the whole multiset M, and use a different method to filter false positives (see Section 2.3.2).*

binomial distribution, which is known to be precise when the number of trials is not too small (e.g., at least 100). Table 2.5 shows some values of μ and corresponding false negative probabilities, using the Poisson approximation.

False positives. The probability that a pair $\{i, j\}$ with $s(i, j) < \Delta$ is reported depends on how far the mean $s(i, j)\tau$ is from $\Delta\tau$. If the ratio $s(i, j)/\Delta$ is close to 1, there is a high probability that the pair will be reported. However, this is not so bad since $s(i, j)$ is close to the threshold Δ. On the other hand, when $s(i, j)/\Delta$ is close to zero we would like the probability that $\{i, j\}$ is reported to be small. Again, we may use the fact that either $f(c(i), c(j))\tau \geq 1$ (in which case the pair is exactly counted and reported if and only if $s(i, j) \geq \Delta/2$). For $s(i, j) < \Delta/2$ we can use Chernoff bounds, or the Poisson approximation, to bound the probability that $M(i, j) > \Delta\tau/2$. Figure 2.2 illustrates two Poisson distributions (one corresponding to an item pair with measure three times below the threshold, and one corresponding to an item pair with measure at the threshold).

2.3.2 Filtering of BiSam output

The BiSam algorithm generates a stream of weighted item pairs that may be very large. In order to obtain a more succinct output we propose a filtering phase that eliminates pairs that are not similar enough. This task can be carried out in at least three ways:

Exact threshold filtering: A weight threshold w can be set, depending on the similarity one is interested in, and can be used in order to filter out those pairs whose sum of weights is below the threshold. As discussed in the previous section this gives an output where false positive and negative probabilities can be rigorously analyzed. This method requires that the filter stores a set of weighted samples M, e.g. using a hash table, keeping track of the current sum for each pair seen. In the I/O model, the best implementation is via sorting of the output produced by IOBiSam. In the standard model where space is a bigger issue, the next methods may offer better guarantees at the cost of a more complex implementation;

Checking similarities: The weight threshold w implies that we filter away those pairs whose similarity is far below w/τ. An alternative is to spend

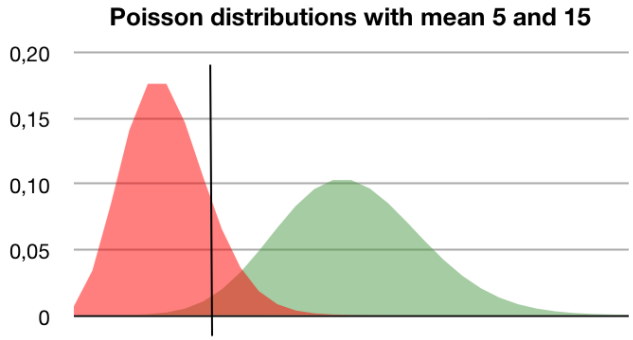


Figure 2.2: *Illustration of false negatives and false positives for $\tau\Delta = 15$. The leftmost peak shows the probability distribution for the number of samples of a pair $\{i, j\}$ with $s(i, j) = \Delta/3$. With a probability of around 13% the number of samples is above the threshold (vertical line), which leads to the pair being reported (false positive). The rightmost peak shows the probability distribution for the number of samples of a pair $\{i, j\}$ with $s(i, j) = \Delta$. The probability that this is below the threshold, and hence not reported (false negative), is around 1.8%.*

more time on the pairs output by BiSam, using a sampling method to obtain a more accurate estimate of $|S_i \cap S_j|$. A suitable technique could be to use min-wise independent hash functions [21, 64] to obtain a sketch of each set S_i. It suffices to compare two sketches in order to have an approximation of the jaccard similarities of S_i and S_j, which in turn gives an approximation of $|S_i \cap S_j|$. Based on this we may decide if a pair is likely to be interesting, or if it is possible to filter it out. The sketches could be built and maintained during the ItemCount procedure using, say, a logaritmic number of hash functions. [64] presents an efficient class of (almost) min-wise independent hash functions.

For some similarity measures such as *lift* and *overlap coefficient* the similarity of two sets may be high even if the sets have very different sizes. In such cases, it may be better to sample the smaller set, say, S_i, and use a hash table containing the larger set S_j to estimate the fraction $|S_i \cap S_j|/|S_i|$. However, this requires that the whole dataset fits in memory.

Most frequent pairs in a stream: This technique consists in the use of a streaming algorithm for finding the pairs whose sum of associated weights exceeds a given user defined threshold. Many algorithms exist that address this problem, see [37] for a comprehensive treatment and an experimental comparison, but only some of them are able to manage weighted items. One algorithm suiting the needs of BiSam well is SpaceSaving [79]. See [80] for a more detailed description of the algorithm. In the following we will describe a modification of SpaceSaving that takes into account the weights of pairs without adding any cost to the computational time.

Weighted SpaceSaving

Here we describe a modification of SpaceSaving that supports weights in the stream M. Such a modification has already been presented in [38], but our approach is different in the sense that using some slackness in the space allowed, we get constant time updates for the underlying data structure. We will refer to our algorithm with the name WSpaceSaving. In the following we consider the elements of M as pairs, but the algorithm works for generic elements.

As pointed out before, we are interested in reporting only those pairs in M whose sum of weights exceeds a certain threshold. Let $N := \sum_{(i,j)\in M} M(i,j)$; given a user defined threshold ϕ we want to report those pairs $p = (i,j)$ whose sum of associated weights $M(i,j) = M(p)$ is larger than ϕN, for some $\phi > 0$. We discuss the choice of ϕ below.

In the following we will call the sum of associated weights $M(i,j)$ of a pair (i,j), the pair's *weight*, and the threshold ϕ the *cut* weight. Moreover, we will denote the associated weight of a pair $(i,j) = p$ with ω.

In order to have the desired result, we maintain a collection of entries, each of which contains a pair, plus an estimate of the weight. The estimate is denoted $count_v$. Moreover, we keep track of the minimum *count* among all the recorded entries, and refer to this value with the name *min*.

The size l of the collection has to be chosen according to the precision of the desired result, since the algorithm can output pairs whose weight is larger than $(\phi - 2/l)N$, and guarantees to output each pair having weight larger than ϕN.

The algorithm works in a fairly simple way: When a new pair (i,j) arrives, we look for it in the collection; if it is already recorded in some position v, we update $count_v$ adding the associated weight ω (which is $\max\{1, f(|S_i|,|S_j|)\tau\}$ in the case of BiSam) to it and we move to the next pair. If the pair is not in the collection we put the pair in the data structure, replacing a pair among the ones having small estimated sum of associated weights. Suppose this pair appears at position v — then we put in that position the pair (i,j), and assign $count_v = count_v + \omega$. Figure 2.2 reports the pseudocode for the updating procedure.

```
1:  procedure WSpaceSaving(M, φ)
2:      while there is a new pair (i, j) from the stream do
3:          if (i, j) is in the collection D at position v then
4:              count_v := count_v + ω
5:          else
6:              Choose a pair from the low weight pairs bucket
7:              Let v be its position
8:              D_v := (i, j)
9:              count_v := count_v + ω
10:         end if
11:     end while
12:     ∀v . v ∈ {1, ..., l} : count_v ≥ φN output D_v
13: end procedure
```

Algorithm 2.2: *Pseudocode for the* WSpaceSaving *algorithm. We remark that, in the case of* BiSam, $\omega := \max\{1, \tau f(|S_i|,|S_j|)\}$

Choosing the pair to replace. We describe how to implement WSpace-Saving in a way that only a constant number of operations are needed in case of an update to the data structure. We maintain the pairs along with their estimated sum of associated weights in buckets, each of which contains pairs with *count* in a certain range. The size of ranges is increasing by a factor of 2, so we only have to keep track of a logarithmic, with respect to N, number of buckets. In particular, we will have the ranges: $[1,2), [2,4), [4,8), \ldots,$ $[2^{k-1}, 2^k), [2^k, \phi N), [\phi N, N]$. In this way, when we need to find a pair whose $count = min$, we can go directly in the nonempty bucket with the lowest weight range, and pick up an arbitrary pair contained in it. Moreover, when it is necessary to move a pair in a new bucket, it is sufficient to move it in the bucket representing the next range, eventually initializing the bucket. These operations can easily be done in constant time per update. Once a bucket is empty it will never receive a new pair again, so we can directly switch to the next one.

For what concerns the correctness of the algorithm, we will first describe some properties.

Lemma 2.3.1. *At any point in time* $\sum_{v \in \{1,\ldots,l\}} count_v = N$

Proof. The lemma can be proved via induction on the length of the stream. The main idea is that at each step, only one counter is incremented with the weight of the new arrived pair. □

Lemma 2.3.2. *Among all counters, the minimum counter value,* min*, is no greater than* N/l*.*

Proof. We can write:

$$\min = l^{-1}\left(N - \sum_{v \in \{1,\ldots,l\}} (count_v - \min)\right);$$

since $\forall v \,.\, count_v \geq \min$, the summation has all nonnegative terms, thus the result. □

Theorem 2.3.1. $\forall (i,j) \,.\, (i,j) \in M \wedge M(i,j) > \phi N \Rightarrow (i,j)$ *is recorded in the data structure.*

Proof. Assume (i,j) do not end up in the data structure; notice that $M(i,j) > min$ at any point in time. Since (i,j) is not in the data structure there has to be a pair that caused the deletion of (i,j) one last time. Since (i,j) has been selected to be deleted, all the pairs in the data structure have to have an estimated frequency larger than ϕN, so $min > \phi N$; by means of Lemma 2.3.2, we also have $min \leq N/l$, so $min > \phi N \geq N/l \geq min$ which is absurd. □

Theorem 2.3.1 states that all pairs (i,j) having frequency $M(i,j) > \phi N$ are reported by the algorithm.

It remains to understand the entity of the error the algorithm introduces. The error depends on the maximum overestimation the algorithm allows. From Lemma 2.3.2 we know that $min \leq N/l$; the pair having $count_v = min$ pertains to a bucket whose range is $[a,b)$. Since $b = 2a \leq 2min \leq 2N/l$ and since we can have overestimated the frequency of a pair using at most b, we get an

additive $2/l-$approximation, hence it is possible that pairs, whose frequency falls in $((\phi - 2/l)N, \phi N]$, are reported.

From the previous theorem, we get the corollary:

Corollary 2.3.1. *The space usage of* BiSam*, when* WSpaceSaving *is used, is* $O(N/(\tau \cdot \Delta))$ *and the computational time remains* $O(mb + \tau \sum_{0<i<j\leq n} s(i,j))$.

Proof. For the time complexity, we have already pointed out that every update to the data structure takes constant time. The space bound is straightforward when we notice that, given the cut weight ϕ, we can have at most $1/\phi$ frequent pairs, so we need at least the space for storing those pairs. For $\phi = (\tau \cdot \Delta)/(2 \cdot N)$, the claim is verified. □

2.4 Experiments

To make experiments fully reproducible and independent of implementation details and machine architecture, we focus our attention on the number of hash table operations, and the number of items in the hash tables. That is, the time for BiSam is the number of items in the input set plus the number of pairs output. The space of BiSam is the number of distinct items (for support counts) plus the space for the filtering algorithm. An exact threshold filter has space usage that is equal to the number of distinct pairs output by BiSam, whereas the most frequent pairs algorithm has space usage that is equal to the output weight of BiSam divided by the weight threshold (see Corollary 2.3.1). Similarly, the time for methods based on exact counting is the number of items in the input set plus the number of pairs in all transactions (since every pair is counted), and the space for exact counting is the number of distinct items plus the number of distinct pairs that occur in some transaction.

We believe that these simplified measures of time and space are a good choice for two reasons. First, hash table lookups and updates require hundreds of clock cycles unless the relevant key is in cache. This means that a large fraction of the time spent by a well-tuned implementation is used for hash table lookups and updates. Second, we are comparing two approaches that have a similar behavior in that they count supports of items and pairs. The key difference thus lies in the number of hash table operations, and the space used for hash tables. Also, this means that essentially any speedup or space reduction applicable to one approach is applicable to the other (e.g. using counting Bloom filters to reduce space usage).

Datasets. Experiments have been run on both real datasets and artificial ones. We have used most of the datasets of the Frequent Itemset Mining Implementations (FIMI) Repository[2]. In addition, we have created three datasets based on the Internet movie database (IMDB). Table 2.6 contains some key figures on the datasets.

2.4.1 Results and discussion

Tables 2.8, 2.9, 2.10 show the results of our experiments for the all_confidence measure. The time and space for BiSam is a random variable. The reported

[2] `http://fimi.cs.helsinki.fi/`

Dataset	distinct items	number of trans.	avg. trans. size	max. trans. size	avg. items support	avg. similarity
Chess	75	3196	37	37	1577	0.3148
Connect	129	67557	43	43	22519	0.1626
Mushroom	119	8124	23	23	1570	0.1523
Pumsb	2113	49046	74	74	1718	0.0120
Pumsb_star	2088	49046	50	63	1186	0.0102
Kosarak	41270	990002	8	2498	194	0.0168
BMS-WebView-1	497	59601	2	161	301	0.0307
BMS-WebView-2	3340	59601	2	161	107	0.0140
BMS-POS	1657	515596	6	164	2032	0.0044
Retail	16470	88162	10	76	55	0.0094
Accidents	468	340183	33	51	24575	0.0248
T10I4D100K	870	100000	10	29	1161	0.0137
T40I10D100K	942	100000	40	77	4204	0.0230
actors	128203	51226	31	1002	12	0.0618
directorsActor	50645	3783	1221	8887	90	0.0978
movieActors	51226	133633	12	2253	33	0.0380

Table 2.6: *Key figures on the datasets used for experiments. The first 13 datasets are from the FIMI repository. The last 3 were extracted from the May 29, 2009 snapshot of the Internet Movie Database (IMDB). The datasets* **Chess**, **Connect**, **Mushroom**, **Pumsb**, *and* **Pumsb_star** *were prepared by Roberto Bayardo from the UCI datasets and PUMBS.* **Kosarak** *contains (anonymized) click-stream data of a hungarian on-line news portal, provided by Ferenc Bodon.* **BMS-WebView-1**, **BMS-WebView-2**, *and* **BMS-POS** *contain clickstream and purchase data of a legwear and legcare web retailer, see [72] for details.* **Retail** *contains the (anonymized) retail market basket data from a Belgian retail store [18].* **Accidents** *contains (anonymized) traffic accident data [50]. The datasets* **T10I4D100K** *and* **T10I4D100K** *have been generated using an IBM generator from the Almaden Quest research group.* **Actors** *contains the set of rated movies for each male actor who has acted in at least 10 rated movies.* **DirectorActor** *contains, for each director who has directed at least 10 rated movies, the set of actors from* **Actors** *that this director has worked with in rated movies.* **MovieActor** *is the inverse relation of* **Actors**, *listing for each movie a set of actors.*

number is an *exact* computation of the expectation of this random variable. Separate experiments have confirmed that observed time and space is relatively well concentrated around this value. The values of τ used are shown in Table 2.7 — they were chosen manually in each case to give a "human readable" output of around 1000 pairs. (For the IMDB dataset Actor and the Kosarak dataset this was not possible; for the latter this behaviour was due to a large number of false positives.) Note that choosing a smaller Δ would bring the performance of BiSam closer to the exact algorithms; this is not surprising, since lowering Δ means reporting pairs having a small value for the similarity measure, increasing in this way the number of samples taken. As noted before, we are usually interested in reporting pairs with high similarity, for almost any reasonable scenario.

The results for the other measures are omitted for space reasons, since they are very similar to the ones reported here. This is because the complexity of BiSam is, in most cases, dominated by the first phase (counting item frequencies), meaning that fluctuations in the cost of the second phase have little effect.

We see that the speedup obtained in the experiments varies between a factor 1.62 and a factor over 36. The largest speedups tend to come for datasets with the largest average transaction size, or datasets where some transactions are very large (e.g. **Kosarak**). However, as our theoretical analysis suggests, large transaction size alone is not sufficient to ensure a large speedup — items also need to have support that is not too small. So while the **DirectorActor** dataset has very large average transaction size, the speedup is not as high as the ones observed for other datasets, because the support of items is low. In a nutshell, BiSam gives the largest speedups when there is a combination of relatively large transactions and relatively high average support. The space results are shown in Table 2.9 and Table 2.10. In particular, Table 2.9 refers to the algorithm when the *Exact threshold filtering* is applied and the space usage ranges from being quite close to the space usage for exact counting, to a decent reduction. Table 2.10 refers to the algorithm when WSpaceSaving is used. In particular, in this case, we are taking into account the version of the algorithm presented in [38], where the space usage would be $N/(\tau \cdot \Delta)$ at the cost of raising the time complexity to $O\left(mb + \tau \sum_{1 \leq i < j \leq n} s(i,j) \log(N/\tau\Delta)\right)$. As can be seen we may get much higher savings in space in this case, up to almost two orders of magnitude for some datasets. Especially, we get large savings for some datasets with many distinct items.

Though we have not experimented with methods based on locality-sensitive hashing (LSH), we observe that our method appears to have an advantage when the number n of distinct items is large. This is because LSH in general (and in particular for cosine similarity) requires comparison of $\binom{n}{2}$ pairs of hash signatures. On the other hand, our algorithm uses time that is n^2 times the average similarity (times a constant τ that is typically small, since we are looking for high-similarity pairs). Table 2.6 shows the average all_confidence similarity of each of our datasets, which is typically 1–2 orders of magnitude smaller than the similarity of the pairs we wish to report.

For the datasets **Kosarak**, **Retail**, **BMS-Webview-2**, **Actors**, and **MovieActors** the ratio between the number of signature comparisons and the number of hash table operations required for BiSam is in the range 9–1340. While these numbers are not necessarily directly comparable, it does indicate that BiSam

has the potential to improve LSH-based methods that require comparison of all signature pairs.

Parameters and output size		
Dataset	τ	#output
Chess	20	986
Connect	20	1008
Mushroom	40	1048
Pumsb	9	844
Pumsb_star	14	1012
Kosarak	6	1710
BMS-WebView-1	30	992
BMS-WebView-2	21	1002
BMS-POS	85	994
Retail	20	1047
Accidents	30	1030
T10I4D100K	40	947
T40I10D100K	30	1087
Actors	8	200445
DirectorsActor	3	714
MovieActors	13	1213

Table 2.7: *Chosen values of parameter τ and the corresponding output sizes.*

Time			
Dataset	BiSam	Exact counting	Ratio
Chess	$1.35 \cdot 10^5$	$22.5 \cdot 10^5$	16.67
Connect	$29.3 \cdot 10^5$	$639 \cdot 10^5$	21.82
Mushroom	$2.08 \cdot 10^5$	$22.4 \cdot 10^5$	10.77
Pumsb	$36.8 \cdot 10^5$	$1360 \cdot 10^5$	36.96
Pumsb_star	$25.4 \cdot 10^5$	$638 \cdot 10^5$	25.12
Kosarak	$108 \cdot 10^5$	$3130 \cdot 10^5$	28.98
BMS-WebView-1	$2.06 \cdot 10^5$	$9.64 \cdot 10^5$	4.68
BMS-WebView-2	$5.66 \cdot 10^5$	$24.4 \cdot 10^5$	4.31
BMS-POS	$35.1 \cdot 10^5$	$246 \cdot 10^5$	7.01
Retail	$15.3 \cdot 10^5$	$80.7 \cdot 10^5$	5.27
Accidents	$115 \cdot 10^5$	$187 \cdot 10^5$	1.62
T10I4D100K	$11 \cdot 10^5$	$62.8 \cdot 10^5$	5.72
T40I10D100K	$42.6 \cdot 10^5$	$841 \cdot 10^5$	19.74
Actors	$144 \cdot 10^5$	$500 \cdot 10^5$	3.47
DirectorsActor	$4688 \cdot 10^5$	$81500 \cdot 10^5$	17.38
MovieActors	$290 \cdot 10^5$	$1070 \cdot 10^5$	3.69

Table 2.8: *Experimental results for all_confidence measure concerning time. The colums ratio represents the savings* BiSam *gets with respect to an exact approach computing all pairs.*

Space (*Exact Threshold Filtering*)			
Dataset	BiSam	Exact counting	Ratio
Chess	$2.20 \cdot 10^3$	$2.66 \cdot 10^3$	1.21
Connect	$4.14 \cdot 10^3$	$6.96 \cdot 10^3$	1.68
Mushroom	$2.92 \cdot 10^3$	$3.65 \cdot 10^3$	1.25
Pumsb	$36.7 \cdot 10^3$	$536 \cdot 10^3$	14.6
Pumsb_star	$44.5 \cdot 10^3$	$485 \cdot 10^3$	10.9
Kosarak	$2306 \cdot 10^3$	$33100 \cdot 10^3$	14.35
BMS-WebView-1	$26.5 \cdot 10^3$	$64.5 \cdot 10^3$	2.43
BMS-WebView-2	$163 \cdot 10^3$	$725 \cdot 10^3$	4.45
BMS-POS	$89.4 \cdot 10^3$	$381 \cdot 10^3$	4.26
Retail	$612 \cdot 10^3$	$3600 \cdot 10^3$	5.88
Accidents	$10.9 \cdot 10^3$	$47.3 \cdot 10^3$	4.35
T10I4D100K	$60.7 \cdot 10^3$	$171 \cdot 10^3$	2.82
T40I10D100K	$168 \cdot 10^3$	$433 \cdot 10^3$	2.57
Actors	$11925 \cdot 10^3$	$32900 \cdot 10^3$	2.76
DirectorsActor	$76104 \cdot 10^3$	$367000 \cdot 10^3$	4.82
MovieActors	$22317 \cdot 10^3$	$55400 \cdot 10^3$	2.48

Table 2.9: *Result of experiments for the all_confidence measure concerning space when the Exact Threshold Filtering is used.*

Space (WSpaceSaving)			
Dataset	BiSam	Exact counting	Ratio
Chess	$2.11 \cdot 10^3$	$2.66 \cdot 10^3$	1.26
Connect	$2.90 \cdot 10^3$	$6.96 \cdot 10^3$	2.40
Mushroom	$2.79 \cdot 10^3$	$3.65 \cdot 10^3$	1.31
Pumsb	$9.33 \cdot 10^3$	$536 \cdot 10^3$	57.48
Pumsb_star	$10.61 \cdot 10^3$	$485 \cdot 10^3$	45.69
Kosarak	$387 \cdot 10^3$	$33100 \cdot 10^3$	85.63
BMS-WebView-1	$7.5 \cdot 10^3$	$64.5 \cdot 10^3$	8.60
BMS-WebView-2	$29.27 \cdot 10^3$	$725 \cdot 10^3$	24.79
BMS-POS	$18.96 \cdot 10^3$	$381 \cdot 10^3$	20.10
Retail	$94.33 \cdot 10^3$	$3600 \cdot 10^3$	38.20
Accidents	$4.75 \cdot 10^3$	$47.3 \cdot 10^3$	9.95
T10I4D100K	$12.52 \cdot 10^3$	$171 \cdot 10^3$	13.65
T40I10D100K	$38.22 \cdot 10^3$	$433 \cdot 10^3$	11.32
Actors	$1731 \cdot 10^3$	$32900 \cdot 10^3$	19.00
DirectorsActor	$58974 \cdot 10^3$	$367000 \cdot 10^3$	6.32
MovieActors	$3461 \cdot 10^3$	$55400 \cdot 10^3$	16.00

Table 2.10: *Result of experiments for the all_ confidence measure concerning space when the version of* WSpaceSaving *presented in [38] is used.*

Chapter 3

Interlude

The BiSam algorithm presented in Chapter 2, looks for all the pairs in a transaction whose associated sampling probability falls in an interval of the form $[r, 1]$. This is actually a concrete example of the more general problem of finding elements belonging to a certain interval of values in a matrix. In this chapter we present a technique for solving the described problem in linear time with respect to the size of the dimensions of the matrix. We will use this technique explicitly in Chapters 5 and 7, with the adaptations necessary to fit the specific settings of those problem. The method can also be applied to the case described in Section 4.3.1, in the paragraph on page 54, where the details of the streaming adaptation are explained.

We think it is useful to give a description that prescinds from the specific incarnation the technique can take for a given problem.

3.1 Problem description

Suppose that we are given two vectors of values $\vec{v} = (v_1, \ldots, v_n) \in U^n$ and $\vec{w} = (w_1, \ldots, w_m) \in U^m$ for a set U with a total order relation $<$ defined on it. Moreover, suppose that the two vectors are sorted in ascending order; i.e, $\forall i, j \in [n],\, i', j' \in [m]\,.\,(i < j \Rightarrow v_i \leq v_j) \wedge (i' < j' \Rightarrow w_{i'} \leq w_{j'})$. Furthermore, suppose that a function g is defined, such that:

$$\begin{array}{ccccccc} g & : & U & \times & U & \rightarrow & I \\ & & u_1 & , & u_2 & \mapsto & g(u_1, u_2) \end{array} .$$

I is a group on which a total order relation $\leq$ is defined. The function g is monotonically non decreasing in both parameters in the group I, that is, modulo the maximum element in I.

The problem we are interested in is finding efficiently the pairs of vector components (v_i, w_j) such that $g(v_i, w_j) \in [p, q] \subseteq I$, where p and q are not necessarily distinct values.

The problem is trivially solved in quadratic time just by checking all the possible pairs of vector components. So solving this problem efficiently means finding an algorithmic way to the result that is linear both in the size of the vectors and in the number of pairs in the output.

In the next section we will present an algorithm that achieves the necessary efficiency.

3.2 Our algorithm

In order to find the pairs we are interested in, we will find some zones in an ideal matrix M that contains all the vector component pairs. We can think of the matrix M associating every row i with the value v_i and every column j with the value w_j. An entry $M_{i,j}$ of M consists of the value $g(v_i, w_j)$. Built in this way, M is monotonic non decreasing both row-wise and column-wise.

Instead of an interval $[p, q]$ we will consider the interval $[p+h, q+h] = [p', q']$, where for each $d \in I$ we have $p' \leq d$. We consider the interval in this form because it simplifies the exposition of the algorithm, but the technique itself is general and can be applied to any interval. We will start looking at the pairs whose associated values appear in the first column, that is, pairs of the form (v_i, w_1). In order to find the first pair that has an associated value belonging to the interval, we look for index i where a *flip* happens. We call *flip* an inversion in monotonicity in a column k, such that $M_{i,k} < M_{(i-1) \bmod n,k}$. For a column with flip position i we notice that: (i) the value in position i, is the smallest; (ii) there is one and only one flip; (iii) all the other values belonging to the interval $[p', q']$ will appear in rows subsequent to the i^{th} having index larger than i modulo n. Index $i_1 = i$ gets marked, since it will be used for the next column. All we need to find now is the row index j, sometimes called *upper boundary* in the remaining part of the Chapter, such that $M_{j,1} \leq q' < M_{(j+1) \bmod n,1}$, and, because of (iii), it suffices to look for this index in rows below the i^{th} in the column. Once found, position $j_1 = j$ gets marked too and the pair of coordinates $\langle i_1, j_1 \rangle$, defining the borders of the zone of values falling in $[p', q']$ in the first column, is output.

Suppose that the algorithm has found $\langle i_k, j_k \rangle$ for column k; it is now necessary to find $\langle i_{k+1}, j_{k+1} \rangle$ in the next column. Therefore, the algorithm starts looking for a flip in column $k + 1$. This search can start from index i_k, that is, from $M_{i_k,k+1}$; until the flip is found, it suffices to look at rows with index smaller than i_k. Again, the row position i_{k+1} where the smaller value in column $k+1$ lies, gets marked. In order to find the last pair that has an associated value belonging to the interval, we look for the row where the largest value smaller than q' lies in the column. The algorithm starts looking for such a value from the row index j_k that has been marked in the preceding column, decreasing the index until the search succeeds. The reason for this choice is that, since g is monotonic, the values $M_{j_k+d,k+1}$, $d \in [1, n - j]$, are either involved in a flip or larger than $M_{j_k+1,k} > q'$. Once the new ending position j_{k+1} is found, it gets marked and the pair $\langle i_{k+1}, j_{k+1} \rangle$ is output for column $k + 1$. If the algorithm reaches row 1 during the search of either i or j in any column, it continues to search starting from the bottom of the column. The values in the columns *wrap up*, so, as it happens for Karnaugh maps [68], the values $M_{1,k}$ and $M_{n,k}$ have to be considered adjacent.

It is important to point out that rows have the same characteristics of columns in terms of behaviour of the values. Therefore, each row is monotonically non decreasing, has one and only one pair of adjacent columns where a flip happens and the index where the flip happens marks the position of the smallest value in the row.

Algorithm 3.1 contains the pseudocode for the described technique, and Figure 3.1 gives a graphical representation of how intervals behave in the matrix M.

```
1: procedure FindInterval(v⃗, w⃗, q′)
2:     s := 0
3:     while g(v_{(s−1) mod n}, w_0) ≤ g(v_s, w_0) do
4:         s := (s + 1) mod n
5:     end while
6:     s′ := s
7:     while g(v_{(s′+1) mod n}, w_0) ≤ q′ do
8:         s′ := (s′ + 1) mod n
9:     end while
10:    o_0 := (s, s′)
11:    for t := 1 to m − 1 do
12:        while g(v_s, w_t) ≥ g(v_{(s−1) mod n}, w_t) do
13:            s := (s − 1) mod n
14:        end while
15:        while g(v_{s′}, w_t) > q′ do
16:            s′ := (s′ − 1) mod n
17:        end while
18:        o_t := (s′, s)
19:    end for
20:    output o⃗
21: end procedure
```

Algorithm 3.1: *Pseudocode for the interval finder. The* **While** *loops on Line 3 and Line 7 find the* flip *and the last value in* $[p', q']$ *respectively in the first column. The* **While** *loop on Line 12 finds the index where the* flip *happens in a generic column* $t > 0$*. The* **While** *loop on Line 15 instead, finds, for a generic column* $t > 0$*, the index* s' *such that* $s(v_{s'}, w_t)$ *is the largest value smaller than* q'*. We do not consider in the pseudocode cases in which there are no interesting values in a column and the patologic case of a column being constant.*

3.3 Analysis

For the first column the algorithm takes at most n steps to find the flip and the upper boundary. As a matter of fact, the loops on Line 3 and Line 7 in Algorithm 3.1 are separated for sake of clarity, but the operations they involve could be carried out together. For subsequent columns it suffices to notice that at most n increases to the row index of the flip position will be performed in all columns, since there can be only one flip per row. The same reasoning applies for the the number of increases to the row index for finding the upper boundary of intervals in all columns. This results in at most $3n$ operations, which is exactly the linear running time we were trying to achieve. Clearly, in order to report the z pairs in the interval, it would suffice to scan the interval itself and output the pairs, which would take z steps.

The total running time is therefore $O(n+m+z)$, when the algorithm outputs, along with the boundaries of the interval, the pairs that the interval contains.

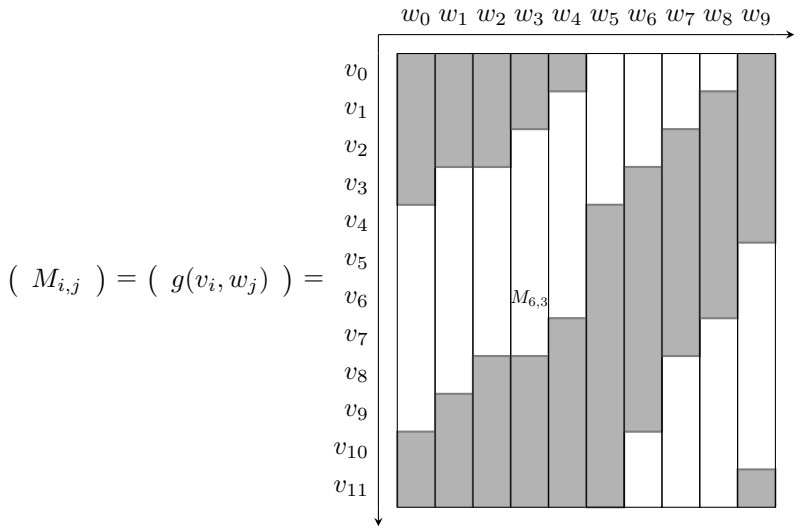


Figure 3.1: *The grey intervals define the zone in the matrix M containing values belonging to $[p', q']$; these values are obtained computing the monotonically non decreasing function g on pairs of vector components; the latter are represented in sorted order, according to an order relation, on the respective axes.*

Chapter 4

BiSam - a Streaming Approach

While there has been a lot of work on finding frequent itemsets in transaction data streams, none of these solve the problem of finding similar pairs according to standard similarity measures. In this chapter we present a first attempt at dealing with this, arguably more important, problem.

We start out with a negative result that also explains the lack of theoretical upper bounds on the space usage of data mining algorithms for finding frequent itemsets: any algorithm that (even only approximately and with a chance of error) finds the most frequent k-itemset must use space $\Omega(\min\{mb, n^k, (mb/\varphi)^k\})$ bits, where mb is the number of items in the stream so far, n is the number of distinct items and φ is a support threshold.

To achieve any non-trivial space upper bound we must thus abandon a worst-case assumption on the data stream. We work under the model that the transactions come in random order, and show that surprisingly, not only is small-space similarity mining possible for the most common similarity measures, but the mining accuracy *improves* with the length of the stream for any fixed support threshold.

4.1 Introduction

The problem we face is very close to the one presented in Chapter 2. As a matter of fact we have a set of m sets ("transactions"), each a subset of $\{1, \ldots, n\}$, and we want to find interesting associations among items in these transactions. This problem is thus the one framed in the "market basket" model where we are interested in finding those pairs of items that are frequently bought together. As pointed out in the previous chapter, understanding whether a pattern is interesting or not must rely on various similarity measures, since the interest of a pattern is problem dependent. We report here the names of some of the most used similarity measures, as we did in Chapter 2, in order for the reader to have a simpler access to the relevant references: *Jaccard* [34], *cosine* and *all_confidence* [74, 84]. Also in this case, besides the cited measures, we are interested in association rules, which are intimately related to the *overlap coefficient* similarity measure. As already pointed out in Chapter 2, [59, Chapter

5] contains background and discussions of similarity measures.

In this chapter, we initiate the study of this problem in the *streaming model* where transactions arrive one by one, and we are allowed limited time per transaction and very small space. The latter constraint implies we cannot hope to store much information regarding pairs that are not similar and, moreover, we cannot store the input. In particular, classical frequent item set algorithms such as A-Priori [6] and FP-growth [60] that work in several passes over the data cannot be used. The survey of Jiang and Gruenwald [66] gives a good overview of the challenges in data stream association mining.

Previous works on transaction data streams have focused on finding frequent itemsets, and can be classified in the following way [100]:

Landmark model: The frequent itemsets are searched for in the whole stream, so that itemsets that appeared in the far past have the same importance as recent ones;

Damped model: This model is also called *Time-Fading*. Recent transactions have a higher weight than the older ones, so nearer itemsets are considered more interesting than the further;

Sliding window: Only a part of the stream is considered at a given time in this model, the one falling in the sliding window. This implies storing information concerning the transactions falling within the window, since whenever a transaction gets out of the window span, it has to be removed from the counts of the itemsets. This approach falls in the more general streaming technique described in Subsection 1.4.2

The last two models make the problem of achieving low space usage simpler, since most of the information in the stream has little or no effect on the mining result. The challenge is instead to handle the real-time requirements of data stream settings.

All the cited approaches look for frequent items and do not try to compute any similarity, relying on the tacit assumption that whatever is frequent is automatically interesting. This assumption is not always true:

Example 4.1.1. Suppose we have item 1 appearing in 20% of transactions, item 2 appearing in 20% of transactions, and the pair $\{1, 2\}$ appears in 10% of transactions. Suppose moreover that the pair $\{3, 4\}$ appears in only 5% of transactions and that these transactions are the only ones in which 3 and 4 appear. The set $\{1, 2\}$ has a frequency that is two times the one of $\{3, 4\}$. But looking at the similarity function *cosine*, we can easily realize that $s(1, 2) = 10/20 = 0.5$ while $s(3, 4) = 5/5 = 1$. If we base the idea of similarity only on frequencies, we are likely to miss the pair $\{3, 4\}$ which holds a much higher similarity than the more frequent pair $\{1, 2\}$.

Notice also that $\{3, 4\}$ holds a higher similarity for *all* the measures we are addressing, so the example shows how frequencies alone do not suffice to infer similarity properties of pairs. ◦

Our contributions. In this chapter we address the problem of finding similar pairs in a stream of transactions. We first show a negative result, which is that

a worst-case stream does not allow solutions with non-trivial space usage: To approximate even the simplest similarity measure one essentially needs space that would be sufficient to store either the number of occurrences of all pairs or the contents of the stream itself. Imposing a minimum support φ for the items we are interested in alleviates the problem only when φ is close to the number of transactions.

Theorem 4.1.1. *Given a constant $k > 0$, and integers m, n, φ, consider inputs of m transactions of total size mk with n distinct items. Let $s_{\max}$ denote the highest support among k-itemsets where each item has support φ or more. Any algorithm that makes a single pass over the transactions and estimates $s_{\max}$ within a factor $\alpha < 2$ with error probability $\delta < 1/2$ must use space $\Omega(\min(m, n^k, (m/\varphi)^k))$ bits in expectation on a worst-case input distribution.* ∘

This lower bound extends and strengthens a lower bound for single-item streams presented in [40].

Of course, many data streams may not exhibit worst-case behavior. Several papers have considered models of data streams where the items are supposed to be independently chosen from some distribution, or presented in random order [41, 95, 26, 58]. We present an upper bound that works for a worst-case set of transactions under the condition that it is presented in random order, which is sufficient to bypass the lower bound. Our method is general in the sense that it can evaluate the similarity of pairs according to several well-established measure functions.

Theorem 4.1.2. *Let $\delta > 0$ be constant, and s, $M > 1$ be integers. We consider a data stream of transactions (subsets of $\{1, \dots n\}$) of maximum size M, where in each prefix the set of transactions appears in random order. For all the similarity measures in Table 2.1 there is a streaming algorithm (depending on s and M) that maintains a "$1\pm\delta$ approximation" of the s most similar high-support pairs in the stream, as follows: Within the m transactions seen so far, let Δ be the sth highest similarity among pairs $\{i, j\}$ where both i and j appear at least φ times, where φ can be any function of m. There exists $L = O(\log(mn))$ such that if $\Delta > \frac{L}{\varphi} \max\left\{\sqrt{\frac{mbM}{s}}, M\right\}$, then the pairs maintained all have similarity at least $(1-\delta)\Delta$ with high probability, and all such pairs with similarity $(1+\delta)\Delta$ or more are reported. To process a prefix of mb items, the algorithm uses time $O(mb \log(nm))$, with high probability, and space $O(n + s)$.* ∘

It is worth noticing that s can be chosen as $O(n)$, which yields a space usage linear in the number of distinct items. Conversely, choosing s smaller does not improve the space usage, so we may assume $s \geq n$. In absence of a known bound on the maximum transaction size, one can use $M = n$. Then the algorithm guarantees to detect pairs with similarity at least $\frac{L}{\varphi} \max\left\{\sqrt{mb}, n\right\}$. Using $s \geq n$ and ignoring the logarithmic factor L this means that up to input size $mb = n^2$ we can detect similarity n/φ, and after this point we can detect similarity $\sqrt{mb}/\varphi$. Assuming that φ is chosen as a linear function of m (relative support threshold), we see that the accuracy improves with the length of the stream.

4.1.1 Previous work

Denote by m the number of transactions seen up to the moment in which we want to report the similar pairs. Let n indicate the number of distinct items that can appear in transactions. Without loss of generality we can assume these items are in the set $\{1, \ldots, n\}$. Parameter b is the average length of transactions (such that mb is the size of the dataset seen so far).

Most of the algorithms we describe actually consider the problem of finding frequent objects in a stream of items, so they do not focus on itemsets, like we do. But given a stream of transactions we can of course generate the stream of all pairs occurring in these transactions, and feed them to a frequent item algorithm. (We do not consider here that this might not be possible for large transactions in settings where real-time constraints are important.)

Landmark model

Many research papers have addressed the problem of frequent items in a stream. Starting from the seminal paper [7] streaming algorithms have started to flow in recent years. Many important contributions to the problem of frequent items (and indirectly frequent itemsets) have thus been presented.

In several independent papers [81, 41, 70] algorithms have been presented that can find all pairs with support at least k using space $|S|/(k-1)$ and constant time per pair in the stream S. These algorithms may generate false positives, i.e., it is only known that the output will contain the frequent pairs.

Cormode and Muthukrishnan [40] consider the problem of reporting *hot items* in a fully dynamic database scenario. The space usage is similar to the schemes above, but the error probability can be reduced arbitrarily (at the cost of space).

Also in [40] is a lower bound on the number of bits of memory necessary in order to answer queries that concern reporting the items with frequencies over a certain threshold. This lower bound is extended and generalized by our lower bound in Theorem 4.1.1.

In [28] the Count Sketch algorithm tackles the problem of reporting the k most frequent itemsets. For worst-case distributions their algorithm has similar performance to those mentioned above, but for skewed distributions they are able to detect itemsets with smaller frequencies in the same amount of space.

A false negative approach

Yu et al. [95] present algorithms directly addressing the problem of finding frequent itemsets in a transaction stream. The algorithm does not find itemsets that are similar by means of measure functions other than support. Under the assumption that items occur independently (which is arguably quite strong, since we are assuming that there may be dependencies resulting in frequent sets) the authors show upper bounds on space usage similar to those of [40]. The performance is tested on artificial datasets where the independence assumption holds. For itemsets of size two (or more) the paper lacks a theoretical analysis of the proposed algorithm, but claims an empirical space usage bounded by m^3/k^3.

Sampling according to the similarity

Our algorithms builds on top of the idea presented in Chapter 2. The sampling technique used in that algorithm is such that pairs are sampled a number of times that is proportional to their similarity. (A more technical explanation can be found in Section 4.3.1 where we improve the sampling procedure to make it suitable for a streaming environment.) The algorithms presented in Chapter 2 have near-optimal running time, when no information on the distribution of similarities are given. As a matter of fact, the running time is linear in the size of the input and output (when there are many pairs of roughly the same similarity). The methods presented are highly general and apply to many measure functions that are linear in the number of occurrences of a pair. However, the method does not directly apply to a streaming setting since it needs two passes over the data.

4.2 Lower bound

There are two naïve approaches to handling k-itemset support counting in a data stream setting: One consists in storing all the transactions seen (possibly trying to compress the representation), and the other one maintains support counts for all k-itemsets seen so far.

Theorem 4.1.1 says that it is not possible to beat the best of these approaches in the worst case (with support threshold $\varphi = 1$). The proof is a reduction from communication complexity:

Proof. The inputs considered for the lower bound have m transactions of size k. Let $n' = \min(n, \lfloor mk/(2\varphi) \rfloor) - 1$ be the largest possible number of items that can appear φ times in $m/2$ transactions, minus 1. We pick an arbitrary set F of n' items, and will form an input stream that consists of two parts:

- In the first $m/2$ transactions we ensure that each item in F appears φ times or more, while no k-subset of F appears. This can be done by putting one item not in F in each transaction.
- In the last $m/2$ transactions we encode information that will require many bits to store, as detailed below.

Consider the first $s = \min(m/2, \binom{n'}{k})$ transactions in the second part. Since $s \leq \binom{n'}{k}$ we can map the numbers $\{1, \ldots, s\}$ to unique k-itemsets in F. In particular, any bit string $x \in \{0,1\}^s$ can be mapped to the unique set of transactions corresponding to the positions of 1s in x. In this dataset, each k-itemset from F appears at most once.

Suppose we have an algorithm that can determine the support of the most frequent itemset within a factor $\alpha < 2$ with probability $1 - \delta$. This implies that, on inputs where no itemset appears more than twice, the algorithm can distinguish (with probability $1 - \delta$) the cases where the most frequent itemset appears once and twice. Given $x \in \{0,1\}^s$ we consider the memory configuration after the algorithm has seen the set of transactions that correspond to x. This can be seen as a "message" that encodes sufficient information on x that allows us to determine if one of the itemsets we have seen appears later in the stream. Lower bounds from communication complexity (see [73, Example 3.22]) tell us

that even when we allow error probability $\delta < 1/2$ the amount of communication to determine whether $x, y \in \{0,1\}^s$ have a 1 in the same position (corresponding to the same k-itemset appearing twice) is $\Omega(s)$ bits in expectation. This means that the memory representation (even if it is compressed) must use $\Omega(s)$ bits. Using the estimate $\binom{n'}{k} \geq \min(\binom{n}{k}, \binom{mk/(3\varphi)}{k}) = \Omega(\min(n^k, (m/\varphi)^k))$ we get the lower bound stated in the theorem. □

Corollary 4.2.1. *Any deterministic algorithm that determines the highest support in a transaction data stream must, after having processed transactions of total size mb, use space $\Omega(\min(mb, n^k))$ bits on a worst-case input.* ∘

4.3 Our algorithm

We now present a new algorithm for extracting similar pairs from a set of transactions using only one pass over the data. The algorithm is approximate, so false negatives and false positives occur. Most of our discussion will concern space usage, but we are also aiming for very low per-item time complexity of the algorithm. In particular, we will not allow anything like iterating through all pairs in a transaction.

The measures we will address are almost the same reported in the Chapter 2, Figure 2.1. The difference stands in the fact that we do not support *lift*. We remark that also in the algorithm presented in this chapter, the measure *Jaccard* can be computed by means of computing the measure*Dice*. Furthermore we recall the reader that finding pairs with an *Overlap Coefficient* over a certain threshold, entails finding association rules with confidence over the same threshold. It is worth pointing out again that the measures we address are all symmetric. This means that we are interested only in looking at pairs (i, j) where $i < j$. For this reason we will use set notation for the pairs, so instead of (i, j) we will write $\{i, j\}$.

Parameters of the algorithm. We recall that φ is the item support threshold, and M is the maximal transaction size. Increasing φ will decrease the minimum similarity the algorithm will be able to spot. M is a characteristic of the transactions, supplied as a parameter to the algorithm. In absence of a known bound on M, one can set $M = n$. The parameter s determines the space usage of the algorithm, which is $O(n + s)$ words.

Notation. In the streaming framework, the total number of transactions is not known. In order to address this issue, we consider sets of transactions, *prefixes*, of the stream of increasing size. Suppose that so far we have seen m transactions $T_1, \ldots, T_m \subseteq \{1, \ldots, n\}$.

The *current* prefix has length 2^t, $t \in \mathbb{N} \cup \{0\}$ when m falls in the interval $[2^t, 2^{t+1})$. Our algorithm maintains counts of all items and stores copies of the counts every time the current prefix changes (that is: Every time the number of transactions seen is two times the length of the current prefix). Each time the current prefix changes, we update our estimate of the most similar pairs, and use this estimate until the next change of current prefix.

The algorithm is based on two pipelined stages: A stream of pairs generation phase and a store and count phase. We will describe the two phases separately,

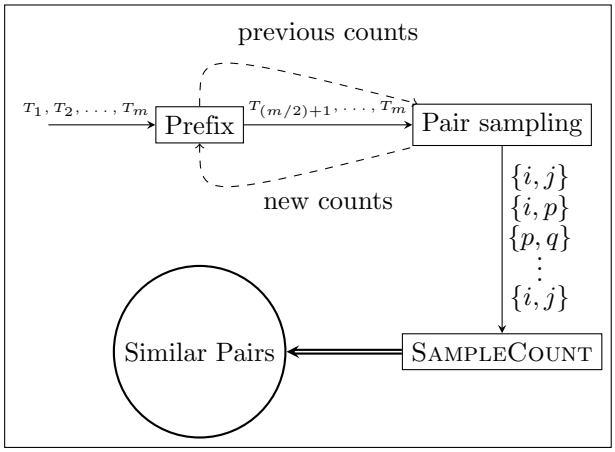


Figure 4.1: *Overview of the algorithm with all its components.*

since the output of the former phase will constitute the input of the latter. Figure 4.1 gives an overview of the algorithm.

The prefixes of the stream are fed to a *pair sampling* stage that uses the stored counts from the previous prefix to compute sampling probabilities. Given the current prefix, the counts relative to that prefix will be used in order to sample pairs in the stream, until a new set of counts is stored for the prefix of length 2^{t+1} The idea is that, since transactions come in random order, the sampling probabilities associated with the pairs in each prefix, should be approximately the same as for the BiSam (Subsection 2.2.1) sampling procedure (which bases the sampling probabilities on exact item frequencies).

In Section 4.4 we show how this technique samples, with high probability, the pairs having a high enough similarity. In fact, we show that a stronger property holds with high probability: Even when we split the stream into κ chunks, each with the same number of transactions, we will sample these pairs sufficiently often in each chunk to reliably estimate their similarity.

4.3.1 Pair sampling

We base our technique on the sampling method of the BiSam algorithm (Chapter 2). For each transaction the pairs are sampled according to their support, such that the pair $\{i, j\}$ is sampled with probability $\tau f(|S_i|, |S_j|)$, where f is a function that depends on the similarity measure considered, and τ is a parameter that is used to control the sampling rate. We fix $\tau = \frac{4\varphi}{M}$, where the number of chunks κ is given by (4.6).

BiSam idea. The idea is that after both i and j have appeared φ times, the expected number of times $\{i, j\}$ is sampled is proportional to $s(i, j)$. Also, the number of samples follows a highly concentrated (binomial) distribution, so the true similarity can be estimated reliably for pairs that are sampled sufficiently often. For any f that is non-increasing in both parameters, the BiSam algorithm performs the sampling in time that is expected linear in the transaction size

plus the number of samples. However, the time to process a transaction may be quadratic with non-negligible probability, which is problematic for application in a streaming context. We refer to the previous chapter for details.

Streaming adaptation. Two things allow us to arrive at a version suitable for streaming:

- While BiSam produces dependent samples, in the sense that the number of times two different itemsets is sampled is not independent, we show how to make the samples produced independent. This will ensure that the number of samples from each transaction is highly concentrated around its expectation.
- The requirement of minimum support φ will ensure that processing of a single transaction takes "linear time with high probability." More precisely: Any set of consecutive transactions with a total of $\log m$ items will require linear time with high probability.

To achieve independence we will change the sampling probabilities by rounding them down to the nearest negative power of 2. This means that the expected number of times $\{i, j\}$ is sampled is no longer exactly proportional to $s(i, j)$, but is changed by a factor $\gamma_{i,j} \in [1, 2]$. However, since the sampling probability is known, which means that $\gamma_{i,j}$ will be constant for any given $\{i, j\}$, we can still use the sample counts to reliably estimate similarity.

Details. For a transaction T_t we can visualise the pairs in $T_t \times T_t$ as a 2-dimensional table, with rows and columns sorted by support, where we are interested in the pairs below the diagonal (index $i < j$). The cost of sorting will be swallowed by the cost of the subsequent operations. Since f is non-increasing the sampling probabilities are decreasing in each row and column. This means that for any $k > 0$, in time $O(|T_t|)$ we can determine what interval in each row of the table is to be sampled with probability 2^{-k}. In order to do this, we just have to maintain references to where the next interval starts. Consider the case when $k = -1$. We can browse the bottom row until we find a pair $\{1, j\}$ such that $f(T_t[1], T_t[j]) < 1/2$. We can mark position $(1, j)$ in order to remember where to start looking for pairs belonging to the next interval. We then go to row 2 and we look at column j; from here we start looking at columns $j - p$, with $p \in \{1, \ldots, j-3\}$ until we find a q such that $f(T_t[2], T_t[q]) \geq 1/2$ and mark the position $q + 1$. This will tell us the position where the rows belonging to the next interesting interval starts on row 2. We then move to row 3 and carry out the same procedure until no more pairs can be found with an associated probability of $1/2$. The total number of pairs scanned is at most $2|T_t|$ and we know in which columns the interval corresponding to the sampling probability 2^{-2} starts. Assume we have found the interval for $k = x - 1$. This means that we know where the interval for $k = x$ starts on all the rows we have scanned so far. Assume y and w are the indexes of rows where the first and the last pair with associated probability 2^{-x} lie. We start from row y and look for the first column index where $f(T_t[y], T_t[r]) < 2^{-x}$. We again mark the position and carry out the same operations described above; that is, we scan row $y + 1$ and start decreasing the column index in order to find the first index where $f(T_t[y], T_t[r]) \geq 2^{-x}$. Also in this case, the number of pairs scanned is at most

$2|T_t|$, since we know, for each row index y, where the interval starts; that is, either the index of the first interesting value or the position $(y, y+1)$. Thus we only need to find where the interval finishes, which means going back to at most column $w+1 \leq |T_t|$ or reaching the diagonal in at most $|T_t| - y$ steps.

It is interesting to notice that the method described is an *ad-hoc* one for a specific determination of a more general and abstract problem. This abstract problem is the one of finding elements belonging to an interval in a set of values; that is, the problem we described in Chapter 3. So, another possible approach would be finding the specific adaptation of the general technique provided there. Chapters 5 and 7 show another case in which the same problem is faced, in distinct frameworks, and using the general technique, with the appropriate adaptations.

To produce the part of the sample for one such interval, we describe a method for producing a random sample of $S = \{1, \ldots, \phi\}$, for a given integer ϕ, where each number is sampled with the same probability p. Since $p\phi$ may be much smaller than ϕ, we want the time to depend on the number of samples, rather than on ϕ. This can be achieved using a simple recursive procedure similar to the one used in efficient implementations of reservoir sampling [91]: With probability $(1-p)^\phi$ we return an empty sample. Otherwise, we choose one random element x from S, and recursively take a sample of the set $S \backslash \{x\}$ with sampling probability p. The set S can be maintained in an array, where sampled numbers are marked. In case more than half of the numbers are marked, we construct a new array containing only unmarked numbers. The amortised cost of the construction of the new array is constant per marking. To select a random unmarked number we sample until one is found, which takes expected $O(1)$ time because no more than half of the numbers are marked. To see why this is true, it suffices to point out that each marked number has at most, and very crudely, probability $1/2$ of being selected. In summary, for each sampling probability 2^{-k} we can compute the corresponding part of the sample in expected time $O(|T_i| + z_k)$, where z_k is the number of samples. This is done for $k = 1, 2, \ldots, 2\log(nm)$. Sampling probabilities smaller than $(nm)^{-2}$ are ignored, since the probability that any such pair would be sampled in any transaction is less than $1/m$. That is, with high probability ignoring such pairs does not influence the sample. To state our result, let $2^{-\mathbb{N}}$ denote the set of negative integer powers of 2.

Lemma 4.3.1. *Let $\tilde{f} : \mathbb{N} \times \mathbb{N} \to 2^{-\mathbb{N}}$ be non-increasing in both parameters. Given a transaction T_t and support counts $|S_i|$ for its items, in expected time $O(|T_t|\log(nm) + z)$ we can produce a random sample of z 2-subsets of T_t such that:*

- *$\{i, j\}$ is sampled with probability $\tilde{f}(|S_i|, |S_j|)$ if $\tilde{f}(|S_i|, |S_j|) > (nm)^{-2}$, and otherwise with probability 0, and*
- *The samples are independent.* ∘

For all similarity measures in Table 2.1 and any feasible value of τ, the minimum support requirement will ensure that the expected number of samples in a transaction is at most $|T_t|$. This means that for each transaction T_t, the time spent is $O(|T_t|\log(nm))$ with high probability.

4.3.2 SampleCount

This phase sees the stream of pairs generated by the pair sampling, and has to filter out as many low similarity pairs as possible, while successfully identifying high similarity pairs. By the properties of pair sampling, this is essentially the task of identifying frequent pairs in the stream of samples. We aim for space usage that is smaller than that of standard algorithms for frequent item mining in a data stream. In order to accomplish this we use a modification of an algorithm presented in [41]. That algorithm finds frequent items in a randomly permuted stream of items, and so does not directly apply to our setting where only the transactions are assumed to come in random order. In [41], the authors are able to sample random elements by simply taking the first elements from the stream. This would not work in our setting, where all these elements might be pairs coming from the same transaction.

Reservoir sampling. Instead, we use a *reservoir sampling* method [91]. We sketch the mechanism here and we refer to the original paper for a complete description. Suppose we have a sequence of d items and we want to sample a random subset of the sequence. We first of all put in the sample the first s elements that we see. For each subsequent element, in position $t > s$, we will put it in the sample with probability s/t. When a new element has to be included in the sample, another one that is already part of the sample has to be evicted. Each element of the set of samples will be chosen as the victim with probability $1/s$. This technique ensures we will end up with a set of samples that is a true random sample of size s.

SampleCount. We consider the stream of pairs divided into κ chunks. The pair sampling generates these chunks such that each chunk corresponds to some set of transactions (i.e., all the pairs sampled from each transaction end up in the same chunk).

We run reservoir sampling on every other chunk to produce a truly random sample of size $s/2$. We then proceed to count the occurrences of the elements of the sample in the next chunk. Assume in the following that we number chunks by $[\kappa]$, such that reservoir sampling is done on even-numbered chunks, indexed by $[\kappa_{\text{even}}]$.

When doing the above, whenever we see a pair $\{i, j\}$ whose count must be updated, we weigh the sample by the factor $\gamma_{i,j}$ that got "lost" during the pair sampling phase, so as to consider an expected number of samples exactly proportional to $s(i, j)$. At the end of a counting chunk we estimate the similarities of all pairs sampled, and keep the $s/2$ largest similarities seen so far. At the end of the stream the similarity estimates found are returned to supersede the previous estimates. Pseudocode for the SampleCount is shown in Algorithm 4.1.

4.4 Analysis

Using the same notation as of Chapter 2, S_i denotes the set of transactions containing the element i. Again this means that $S_i \cap S_j$ is the set of transactions containing the pair $\{i, j\}$. Let S_i^1 denote the set of transactions containing i in

```
1: procedure SampleCount(P, s, size)
2:     S'_out := ∅
3:     while There are elements in P do
4:         S' := ∅
5:         S := ∅
6:         S := the first s/2 elements in P
7:         t := s/2
8:         while t < size do
9:             i := the next element in P
10:            t := t + 1
11:            Choose uniformly at random a number r ∈ [0, 1]
12:            if r ≤ s/(s/2 + t) then
13:                Choose a victim j uniformly at random from S
14:                Substitute j with i
15:            end if
16:        end while
17:        Initialize(S', S)
18:        while (t < 2 · size) do
19:            i := the next element in P
20:            t := t + 1
21:            if i ∈ S then
22:                S'(i) := S'(i) + γ_i
23:            end if
24:        end while
25:        S'_out := the s topmost distinct items between S'_out and S'
26:    end while
27:    output S'_out
28: end procedure
```

Algorithm 4.1: *Pseudocode for the* SampleCount *phase. The input parameter P, it is a stream of pairs, each of which has associated a similarity value. The length of P is known. S' on Line 17, is an associative array indexed on the distinct items present in S; initializing it means putting all its entries to 0.*

the current prefix of the stream. Similarly, S_i^k will denote the set of transactions containing i in C_k, the chunk k of the suffix of the stream up to the point in which a new current prefix changes the counts of items occurrences. So $S_i^k = S_i \cap C_k$

Definition 4.4.1. *Given $x, y \in \mathbb{R}$ we say that x (δ, L)-approximates y, written $x \overset{\delta,L}{\simeq} y$, if and only if $x \geq L$ implies $x \in [(1-\delta)y; (1+\delta)y]$.* ∘

The notation extends in the natural way to approximate inequalities.

In what follows we will use (δ, L)-approximations, where $L = C\log(mn)$ for a suitably large constant C (depending on the accuracy δ in Theorem 4.1.2). The task is to analyze the accuracy of the new approximation computed when the current prefix changes. We introduce two random events, GoodPermutation (GP) and GoodBisamSample (GBS), and bound the probability that they do not happen.

A permutation of the transactions is called *good* for $\{i, j\}$, denoted $\mathrm{GP}_{i,j}$, if and only if the following conditions hold (for the current prefix):

1. $|S_i^1| \overset{\delta,L}{\simeq} |S_i|/2$ and $|S_j^1| \overset{\delta,L}{\simeq} |S_j|/2$;
2. $\forall k \,.\, |S_i^k \cap S_j^k| \overset{\delta,L}{\simeq} |S_i \cap S_j|/2k$;

Essentially, goodness means that the frequencies of individual items are close in the first and second half of the current prefix and the frequency of the pair is evenly spread over the chunks in the second part of the current prefix.

Lemma 4.4.1. *Given $\delta \in [0; 1] \subseteq \mathbb{R}$, we have:*

$$\Pr[\mathrm{GP}_{i,j}] \geq 1 - 6 \cdot e^{\frac{-|S_i|\delta^2}{6}}$$

Proof. An interesting property of the random variables $|S_i^1|$ and $|S_i^k \cap S_j^k|$ is that they are negatively dependent [43]. First of all we bound the probability that $|S_i^1|$ is far from $|S_i/2|$. Using Chernoff bounds we can write:

$$\Pr[||S_i^1| - |S_i|/2| \leq \delta|S_i|/2] \leq 2 \cdot e^{-\frac{|S_i|\delta^2}{6}} \tag{4.1}$$

Looking at $|S_i^k \cap S_j^k|$ we can write:

$$\Pr[|S_i^k \cap S_j^k| - |S_i \cap S_j|/2\kappa| \leq \delta|S_i \cap S_j|/2\kappa] \leq 2 \cdot e^{-\frac{|S_i \cap S_j|\delta^2}{6\kappa}} \tag{4.2}$$

We use the fact that Chernoff bounds also hold for negatively dependent random variables. Since the last bound is the weakest, the lemma follows. □

We want $\mathrm{GP}_{i,j}$ to hold with probability $1 - o(1/n^2)$ whenever items i and j both have support φ. From Lemma 4.4.1 we get that this holds if $|S_i \cap S_j| > C\kappa \log n$, for some constant C (depending on δ). If $s(i,j) > 2\kappa L f(\varphi, \varphi) \geq \kappa L/\varphi$ then $|S_i \cap S_j| \geq 2\kappa L$. Hence, a sufficient condition for the similarity is

$$s(i,j) > \kappa L/\varphi \,. \tag{4.3}$$

It remains to understand what is the probability that, given a good permutation, the pair sampler will take a number of samples for a given pair in each

chunk k that leads to a $(1 \pm \delta)$-approximation of $s(i,j)$. We denote the latter event by $\mathrm{GBS}_{i,j,k}$, and want to bound the quantity $\Pr[\mathrm{GBS}_{i,j,k}|\mathrm{GP}_{i,j}]$.

For this purpose consider the random variable $X_{i,j,k}$ defined as the number of times we sample the pair $\{i,j\}$ in chunk k. Assuming $\mathrm{GP}_{i,j}$ we have that (over the randomness in the pair sampling algorithm and because of the properties of BiSam) $\mathrm{E}[X_{i,j,k}] \overset{\delta,L}{\simeq} \tilde{f}(|S_i^1|,|S_j^1|)\tau|S_i \cap S_j|/2\kappa$. Since the occurrences of $\{i,j\}$ are independently sampled, we can apply a Chernoff bound to conclude $X_{i,j,k} \overset{\delta,L}{\simeq} \mathrm{E}[X_{i,j,k}]$. This leads to the conclusion:

Lemma 4.4.2. $X_{i,j,k} \overset{\delta,L}{\simeq} \tilde{f}(|S_i^1|,|S_j^1|)\tau|S_i \cap S_j|/2\kappa$ ∘

Suppose that $X_{i,j,k}$ is close to its expectation. Then we can use it, with $(1 \pm \delta)$-approximations of $|S_i|$ and $|S_j|$, to compute a $(1 \pm O(\delta))$-approximation of $s(i,j)$. This follows by analysis of the concrete functions f of the measures in Table 2.1 on page 25.

A sufficient condition on the similarity needed for a $(1 \pm \delta)$-approximation of $X_{i,j,k}$ can be inferred from Lemma 4.4.2. in order to get the right approximation for $X_{i,j,k}$, we need to enforce that $\tilde{f}(|S_i^1|,|S_j^1|)\tau|S_i \cap S_j|/2\kappa \geq L$. We know that $\tilde{f}(|S_i^1|,|S_j^1|) \geq f(|S_i^1|,|S_j^1|)/2$. Moreover $|S_i^1|$ and $|S_j^1|$ approximate $|S_i|/2$ and $|S_j|/2$ respectively. It is important to point out that for all the measures we address, $f(|S_i|/2,|S_j|/2) = 2f(|S_i|,|S_j|)$. Thus, overloading the symbol δ we can write: $\tilde{f}(|S_i^1|,|S_j^1|)\tau|S_i \cap S_j|/2\kappa \overset{\delta,L}{\geq} f(|S_i|,|S_j|)\tau|S_i \cap S_j|/2\kappa = s(i,j)\tau/2\kappa$. If $s(i,j) \geq 4\kappa L/\tau$ then $\mathrm{E}[X_{i,j,k}] \geq s(i,j)\tau/4\kappa \geq L$. So it suffices to enforce:

$$s(i,j) \geq 4\kappa L/\tau \; . \tag{4.4}$$

In order to have $O(mb)$ pairs produced by the pair sampling phase, we will choose $\tau = 4\varphi/M$. The expected number of pair samples from T_t is less than $|T_t|^2\tau f(\varphi,\varphi)$, using that f is decreasing. For all measures we consider, $f(\varphi,\varphi) \leq 1/\varphi$, so $|T_t|^2\tau f(\varphi,\varphi) \leq |T_t|^2/M \leq |T_t|$.

It remains to understand which is the probability that a pair of items, each with support at least φ, is not sampled by SampleCount. Let the random variable $X_{.,.,k}$ represent the total number of samples taken in chunk k. The probability that a $\{i,j\}$ is sampled in chunk k is $X_{i,j,k}/X_{.,.,k}$, so the probability that it does not get sampled in any (even-numbered) chunk is $\prod_{k\in[\kappa_{\mathrm{even}}]}(1-X_{i,j,k}/X_{.,.,k})^s$. We have seen before that $X_{i,j,k} \overset{\delta,L}{\geq} s(i,j)\tau/4\kappa$. For what concerns $X_{.,.,k}$ using a Chernoff bound we can get: $X_{.,.,k} \overset{\delta,L}{\simeq} \mathrm{E}[X_{.,.,k}] \leq mb/\kappa$, using the linear upper bound on the number of samples. So we can compute:

$$\prod_{k\in[\kappa_{\mathrm{even}}]}(1-X_{i,j,k}/X_{.,.,k})^s \leq \left(1-\frac{s(i,j)\tau\kappa}{2\kappa\gamma_{i,j}mb}\right)^{s\kappa/2}$$
$$\leq \left(1-\frac{s(i,j)\tau}{4mb}\right)^{s\kappa/2} \leq C\exp\left[-\frac{s(i,j)\tau s\kappa}{8mb}\right]$$

In order for this probability to be small enough ($O(1/m^2)$), we need to bound the similarity to

$$s(i,j) \geq \frac{8mbL}{s\kappa\tau} \tag{4.5}$$

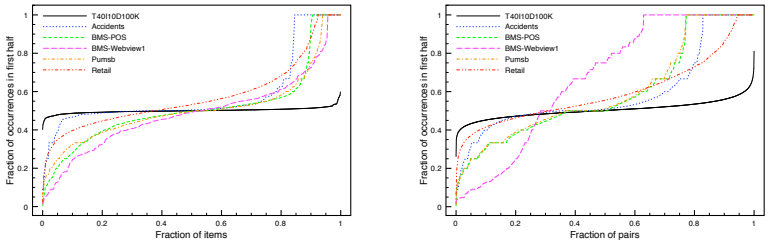


Figure 4.2: *Plots of the ratios* $|S_i^1|/|S_i|$ *and* $|S_i^1 \cap S_j^1|/|S_i \cap S_j|$.

To choose the best value of κ we balance constraints (4.3) and (4.5), getting:

$$\frac{\kappa L}{\varphi} = \frac{mbL}{s\kappa\tau} \Rightarrow \kappa = \sqrt{\frac{mbM}{s}} \tag{4.6}$$

From which we can deduce:

$$s(i,j) = \frac{L}{\varphi} \max\left\{\sqrt{\frac{mbM}{s}}, M\right\} . \tag{4.7}$$

4.5 Dataset characteristics

We have computed, for a selection of the datasets hosted on the FIMI web page[1], the ratios between the number of occurrences of single items and pairs in the first half of the transactions and the total number of occurrences of the same items or pairs. The values of some of this ratios, the most representative, are plotted Figure 4.2; on the x-axis items or pairs are spread evenly, after they have been sorted according to their associated ratio. The y-axis represents the value of the ratios. We have taken into account only items and pairs whose support is over 20 occurrences in the whole dataset, in order to avoid the noise that could be generated by very rare elements. As we can see, the number of occurrences and co-occurrences are not so far from what would be expected under a random permutation of the transactions. The synthetic dataset behaves exactly like we would expect under a random permutation, with the ratio being very close to 1/2 for almost all items/pairs.

This means that even for real datasets, where the order of transactions is not random, the sampling probabilities used in the pair sampling are reasonably close to the ones that would be obtained under the random permutation assumption.

4.6 Conclusions

We presented the first study concerning the problem of mining similar pairs from a stream of transactions that does rely on the similarity of items and not only on the frequency of pairs. A thorough experimental study of (carefully engineered versions of) the presented algorithm remains to be carried out.

[1] `http://fimi.cs.helsinki.fi/`

Chapter 5

Frequent Pairs in Data Streams: Exploiting Parallelism and Skew

We introduce the Pair Streaming Engine (PairSE) that detects frequent pairs in a data stream of transactions. Our algorithm provably finds the most frequent pairs with high probability, and gives tight bounds on their frequency. It is particularly space efficient for skewed distribution of pair supports, confirmed for several real-world datasets. Additionally, the algorithm parallelizes easily, which opens up for real-time processing of large transactions. Our simple Java implementation processes almost 100 million pairs per second on a workstation.

Furthermore we show that any algorithm that returns the exact most frequent pair in a data stream must keep track of the frequencies of all pairs, meaning that an exact approach is infeasible for data sets with many distinct items.

Our theoretical findings are backed by extensive experiments showcasing precision and recall of our method. In particular, we find that our method achieves much better precision than guaranteed by the theoretical analysis, often returning identical upper and lower bounds on the supports of the most frequent pairs.

5.1 Introduction

We have already discussed in this work that a fundamental tasks in knowledge discovery in databases is the mining of high quality association rules from transactional databases over a set of items. The pioneering A-Priori [6] algorithm proposed about two decades ago has paved the way for many important contributions to the problem. Algorithms with much better space and time complexity have since been proposed [20, 60, 86, 88] and showed to efficiently handle large amounts of data.

In this chapter we concentrate again on discovering frequent *pairs*, or $2-$itemsets. It is worth pointing out again that, in our setting, we consider a pair $p = (i, j)$ as a set $\{i, j\}$; therefore we can focus our attention only on pairs such that $i < j$.

Our algorithm can be generalized in a straightforward way to $k-$itemsets, but in that case the analysis becomes more complex. Also, it has been observed that already the case of $2-$itemsets captures the main challenge of frequent itemset mining: *"... the initial candidate set generation, especially for the large $2-$itemsets, is the key issue to improve the performance of data mining"* [86].

5.1.1 Mining data streams

The considerations in this section are vastly similar to the ones already expressed in Section 4.1. We think it is useful for the reader to be able to have a picture of the framework without the need of jumping back to where these notions were firstly exposed. Classical approaches such as A-Priori [6] and FP-growth [60] require several passes over the transactional database and thus it is necessary to have access to a storage system containing the database. As observed by Manku and Motwani [76] this requirement is not practical for many real life applications where we want to mine frequent patterns in only one pass from a high speed stream of transactions. Since this seminal work, many researchers have considered the special requirements of data stream association mining. We refer the reader to the survey of Jiang and Gruenwald [66] for an introduction to this area, and references to many central works. In this paper we restrict our attention to the fundamental case of mining frequent pairs over the entire stream ("landmark model", in the classification of [100]).

We show in Section 5.5 that even if we are only interested in the single most frequent pair, any streaming algorithm that *always* returns the correct pair must (in a certain technical sense) keep count of the number of occurrences of every pair, the cost of which may be prohibitive for datasets with many distinct items. For this reason we concentrate on algorithms that succeed with high probability, and return upper and lower bounds on the number of occurrences rather than precise counts. For example, in the `webdocs` dataset there are around 700 million distinct pairs of items, and keeping all their counts in a hash table would require at least 8 GB of memory. In contrast, we obtain accurate results using a sketch data structure of a few megabytes that fits in L2 cache.

5.1.2 Related work

Heuristic algorithms. Manku and Motwani [76] first recognized the necessity for efficient algorithm targeted at frequent itemsets in transaction streams. They generalized their StickySampling algorithm to a heuristic for transaction streaming and showed empirically that it reliably estimates the frequency of the most frequent itemsets on several benchmarks. The basic idea is to process the dataset in memory-sized chunks, mining each chunk for frequent itemsets to determine which itemsets should be counted in the next chunk. However, this method is vulnerable to large itemsets that are "temporarily frequent". An itemset of size k that is frequent in a chunk will have all its subsets counted in the following chunk, using space 2^k. For this reason it does not seem suitable for general use.

Reduction to the single-item case. Another approach to mining of frequent pairs (mentioned, but dismissed, in [76] and in Chapter 4) is to reduce the problem to that of mining frequent items, which is well-studied in a data stream

context. For a transaction $T \subseteq [n]$, this approach generates all $\binom{|T|}{2} = \Theta(|T|^2)$ pairs of T and feeds the resulting stream S, where the items of S are the pairs generated, into a frequent items algorithm. Let F_2 denote the length of the stream generated in this way. It is known that using space s one can compute the frequency of items (which are in fact pairs in our case) with an additive error of F_2/s [41, 70, 81].

This means that all pairs with frequency above F_2/s can be reported, with computed upper and lower bounds on the frequency that differ by at most F_2/s. While this is optimal over a worst-case data stream where all pairs occur with frequency about F_2/s, some methods, notably the SpaceSaving algorithm [79] (a more detailed version of the paper is [80]), have been observed to produce even tighter bounds on the highest frequencies in practice. However, to our best knowledge, SpaceSaving and related algorithms have never been experimentally investigated in the context of finding frequent itemsets.

Frequent items algorithms aim for using small time per item, and as a matter of fact, the best methods use constant time per item; therefore the time usage for the whole stream is $O(F_2)$. The most space-efficient methods do not parallelize efficiently, as they rely on a single data structure, any part of which may be updated for a particular transaction. Our algorithm, in contrast, parallelizes efficiently without any need for shared memory.

Muthukrishan and Cormode [39] considered finding frequent items space-efficiently in a stream that is highly skewed (Zipfian distribution with parameter greater than 1). In the cited paper they are able to improve over previous results the space needed to identify the most frequent items. However, when considering the stream of all pairs, none of the datasets that we studied exhibits large enough skew for their result to apply.

Algorithms for random streams. Yu et al. [95] presented another algorithm for transaction stream mining. The main idea in their approach is to keep a list of potentially frequent itemsets, and to update the list in a clever way when advancing the stream. The paper contains theoretical bounds for the quality of their estimates. However, in order to derive these bounds, they need to assume that transactions are generated *independently at random* by some process, and their analysis crucially depends on Chernoff bounds that can be used because of this assumption.

In Chapter 4 the assumption is weaker, and is that the *order* of the transactions in the stream is random.

Parallel and distributed algorithms. Since our data streaming algorithm is parallel, it is natural to compare it to the literature on parallel and distributed association mining (see e.g. Zaki's 1999 survey [97]). Often, algorithms in this literature store, partition, and repeatedly access data, so are not viable in a data streaming context. For this reason, our technique for parallelizing the approximate counting of pairs seems novel, and might be applicable to pursue speedups in standard parallel models. In particular, all methods in [97] use either:

- a vertical layout of the data, where for each item a sorted list of of the transactions where it appears is maintained, or

- a shared data structure.

In the former case space efficiency cannot be achieved, since computing the vertical representation entails storing all transactions. In the latter case, concurrency over the shared data structure would find a bottleneck when scaling to many cores. Our algorithm requires neither the former nor the latter technique.

5.1.3 Our contribution

The results presented in this chapter are:

- An efficient algorithm with rigorously understood time and space complexity and output quality; it is analyzed under the assumption of Zipfian distribution of the frequency of pairs; we show that this assumptions holds for many real-life datasets;

- A lower bound on the space that any algorithm has to use when the goal is to report the pair with the highest frequency, or even just the highest frequency seen in the stream of pairs.

The algorithm of the former point is randomized and returns with high probability a correct estimate of the frequency of the most frequent pairs. We build upon well-known streaming algorithms and show how to extend them to transaction streaming.

The complexity as well as the quality of the output is determined by the Zipfian distribution parameters and the space allowed. The space usage is a user-defined parameter from which we will derive bounds for the frequency of pairs which can be detected with high probability as well as on their estimates.

We show through extensive experiments on real and synthetic datasets that in many cases our algorithm performs considerably better than the theoretical analysis suggests.

5.2 Notation

For ease of exposition, in this chapter the notation is slightly different with respect to Chapters 2 and 4. Therefore we specify again all the necessary notation, even if it partially overlaps with what has already been defined previously.

The transaction stream is denoted by $S = T_1, .., T_m$ where $T_i \subseteq [n] = \{1, \ldots, n\}$. Again a subset $p = \{i, j\} \subset [n]$ is called a pair. The set of pairs is denoted by $\mathcal{P}$, while the number of distinct pairs occurring in the stream S is represented using $d \leq \binom{n}{2}$. Furthermore, f will be the number of frequent pairs, where the meaning of *frequent* will be specified in the given context.

The *support* of a pair p is the number of transactions containing p: $\sup(p) = |\{T_j : p \subseteq T_j\}|, 1 \leq j \leq m$.

A hash function $h : \mathcal{P} \to [k]$ for $k \in \mathbb{N}$ is $t-$wise independent if and only if: (i) $\forall p \in \mathcal{P}, c \in [k]\,.\, \Pr[h(p) = c] = 1/k$; (ii) $\Pr[h(p_1) = c_1 \wedge h(p_2) = c_2 \wedge \cdots \wedge h(p_t) = c_t] = k^{-t}$ for distinct pairs p_i, $1 \leq i \leq t$, and $c_i \in [k]$.

The Zipfian distribution with parameters C and z is defined as $f_i = C/i^z$ for the frequency f_i of the ith most frequent pair. Note that we will consider z to be a constant but C does not need to be constant.

5.3 Our approach

5.3.1 Background and intuition

Before formally describing our algorithm let us give some technical background and intuition. An algorithm detecting the frequent items in an item stream can be generalized in a straightforward way in order to find frequent pairs in a stream of transactions: Simply generate for each transaction all subsets of size 2 and treat these subsets as items. In particular, the two well known algorithms Count-Sketch and SpaceSaving could be generalised as described.

In Count-Sketch [28] every item i is hashed by a hash function $h : [n] \rightarrow [k]$ to a bucket B containing a counter c_B. Upon arrival of an item i the corresponding counter is updated by a uniform sign hash function $s(i)$ evaluating i to either 1 or -1. After processing the stream the frequency of a given item i can be estimated as $c_B \cdot s(i)$ where $B = h(i)$. The underlying idea is that the contribution from other items will cancel out. Both h and s are pairwise independent and this is sufficient to show that for an appropriate number of buckets the algorithm produces good estimates where the error is measured with respect to the $2-$norm of the item frequencies. For skewed distribution of the stream frequencies this gives high quality estimates of the heaviest pairs. One can amplify the probability for correct estimates by working with more than one hash function. Upon a query for the frequency of a given item Count-Sketch returns the median of the estimates, i.e., the counters in the buckets the item hashes to.

The SpaceSaving algorithm [79] has been already cited in Chapter 2, where we extended it in order to satisfy the needs of the setting presented there. We sketch again its mechanism here, for the sake of clarity and to make the chapter more readable. We remark that SpaceSaving offers upper and lower bounds on the frequency of the items it reports, rather than an unbiased estimator. It keeps a list of ℓ triples $(\text{item}_j, \text{count}_j, \text{overestimation}_j)$, $1 \leq j \leq \ell$. Until there are free slots amongst the ℓ, it inserts a new triple as $(i, 1, 0)$ when an item i, that is not already stored in one of the slots, arrives. The ℓ triples are sorted according to their *count* value. On arrival of new item i the algorithm checks if it is already in the list, in which case it increases the corresponding counter by 1 and updates the order in the list. Otherwise, since there is no room left, the algorithm replaces the triple $(\text{item}_\ell, \text{count}_\ell, \text{overestimation}_\ell)$ having minimum count with a new triple $(\text{i}, \text{count}_\ell + 1, \text{count}_\ell)$. The intuition is that heavy items will either get early on the "pull position" and won't be evicted from the list, or they will have many chances of entering the data structure and start climbing towards the topmost positions as they appear in the stream. Skewed data will thus get an accurate estimation.

Our algorithm can be seen as a twofold refinement of the above direct approach:

1. In order to address the issue of having a quadratic number of pairs in each transaction, hence a quadratic number of hash values to produce, we exploit parallelism. In this way we are able to distribute the computation among several cores in a way such that each core efficiently computes the pairs hashing to a given subset of the hash table.

2. Assuming Zipfian distribution we want to use the fact that the most fre-

quent pairs will not collide and thus we keep track of the most frequent pair hashing to a given bucket. We will use an important property of SpaceSaving; this property is described in [79] and consists in the fact that in a stream of items, an item having relative frequency at least 1/2 will end up in the first position of the SpaceSaving data structure.

Moreover, SpaceSaving has the nice property to correctly estimate the frequency for the most frequent items for skewed distribution. This is more of a heuristic property since the correctness of SpaceSaving depends on the items order of arrival. However, for certain datasets we are able to obtain very accurate estimates. Details follow.

5.3.2 Our algorithm

The skeleton of our algorithm is the following:

- Hash each pair to a bucket.
- Keep track of the most frequent pair in each bucket.
- Return an estimate of the frequency of the most frequent pair for each bucket.

In the parallel version, each processor keeps track of an interval of the hash table, and the total space remains fixed. Thus, we are in a *shared nothing* model with no need for a shared memory – the only requirement is that each processor sees the input stream. It is well-known that this kind of parallel algorithm scales extremely well compared to algorithms that rely on interprocess communication or shared data structures. Even for the largest datasets that we looked at, it is feasible to keep the entire hash table in L2 cache of the involved processors on a large workstation, resulting in extremely fast processing.

A crucial property in our analysis and experimental evaluations is that most frequent pairs do not collide, and thus we obtain high quality estimates on their frequency. We combine two different ways for estimating the frequency of the heaviest pairs based on the Count-Sketch and SpaceSaving algorithms. In particular, we use a distribution hash function $h : [n] \times [n] \to \{0, \ldots, k-1\}$ to split the set of pairs into k parts, and use a SpaceSaving sketch on each part. The size k of the hash table and the size of the SpaceSaving sketch determines the accuracy of the sketch.

Pseudocode for our algorithm is shown in Algorithms 5.1 and 5.2. PairSE(i, j) produces a table with SpaceSaving and Count-Sketch structures for those pairs (u, v) that have $h(u, v) \in [i, j)$ for some range $[i, j)$.

Parallelizing processing of pairs. Naïvely we could just iterate through all pairs of each transaction T_t, but we would like an algorithm that runs in linear time when the number of pairs hashing to $[i, j)$ is small. This will allow us to split the task of computing the sketch among several cores, all the way to the point where each core processes a transaction in linear time. In other words, given sufficient parallelism we can handle a given data rate even if the transactions are huge.

To achieve this, we exploit the special structure of our hash function: $h(u, v) = (h_1(u)+h_2(v)) \bmod k$ for pairwise independent hash functions $h_1, h_2 : [n] \to [k]$.

To find the pairs in transaction T_t that hash to $[i, j)$ we first sort the sets $H_1 = h_1(T_t)$ and $H_2 = h_2(T_t)$. Pairs with hash value in the right range correspond to elements of H_1 and H_2 with sum in $[i, j) \cup [k + i, k + j)$. We will sometimes denote $[k + i, k + j)$ using $[i, j) + k$. One way to find these would be to iterate over elements of H_1, and for each do two binary searches over the sorted list H_2 to find the values in the right ranges. However, this can be further improved by processing H_1 in sorted order, and exploiting that the intervals of H_2 values to consider will be moving monotonically left. This brings down the time to $O(|T_t| + d[i, j))$, where $d[i, j)$ is the number of pairs hashing to $[i, j)$.

In order to improve the algorithm's accuracy we may run $t = O(\log \frac{1}{\delta})$ copies of the algorithm in parallel.

At the end a pair is reported frequent if it has "won" in at least $t/2$ of its corresponding SpaceSaving data structures. Our experimental results will be for a single run, so the reported accuracy can be improved at the cost of time and space.

The second estimate of the algorithm is based on the Count-Sketch algorithm by Charikar, Chen and Farach-Colton [28]. Here, we have a counter serving as an unbiased estimator for the frequency of the heaviest pair, where *unbiased* means that the estimate does not depend on the order of arrival of pairs.

As in the original Count-Sketch algorithm we will work with an additional pairwise independent hash functions: The sign function $s : \mathcal{P} \to \{-1, 1\}$. With each bucket B we associate a counter c_B. The counter serves the same purpose as in the original algorithm [28].

Upon arrival of a new pair p we update the corresponding bucket, we abuse notation and denote it as $h(p)$, as follows: $c_{h(p)} = c_{h(p)} + s(p)$. The intuition is that the heaviest pair will contribute with the same sign, and contributions from other pairs will cancel out. At the end the algorithm returns $s(p) \cdot c_{h(p)}$ as estimated frequency for the pair where p is the first pair in the SpaceSaving data structure. As we show in the next section, if $\sup(p) > m/2$, then a high-quality estimation of p's frequency is returned.

In order to reduce the error we work again with our simulation of $t = O(\log \frac{1}{\delta})$ independent hash functions and report the median of the results.

Parameters. We will assume that data are skewed and $z > 1/2$. We will distinguish between the cases when $1/2 < z < 1$ and $z > 1$. In order to keep the presentation concise the particular case $z = 1$ will not be analyzed.

In our analysis we will also use as a parameter the number of distinct pairs d occurring in the transaction stream. Of course, the value of d is not known in advance. One can either be conservative and assume $d = \Omega(\binom{n}{2})$, or use efficient methods for estimating the number of occurring pairs, like the one presented in Chapter 7, if two passes over the transaction stream are allowed.

5.4 Analysis

In the following we give a separate analysis of the estimates based on SpaceSaving (giving guarantees on the upper/lower bounds), as well as the unbiased Count-Sketch estimator returned by our algorithm. Note that while Count-Sketch is theoretically superior as it always returns an unbiased estimator, it

```
1: procedure PairSE(Stream S, Interval [i, j), Integer k)
2:     for B ∈ {0, ..., k − 1} do
3:         SpSk[B].Initialise()                ▷ Initialisation of the k buckets
4:     end for
5:     for T ∈ S do
6:         Hash(T, [i, j), SpSk, k)
7:     end for
8:     for B ∈ {0, ..., k − 1} do
9:         (p, c, ϵ) := SpSk[B][0]
10:        c_B := SpSk[B][2]
11:        output (p, c, ϵ)
12:        output c_B · s(p) as an unbiased estimator of the frequency of p
13:    end for
14: end procedure

15: procedure Hash(Transaction T, Interval [i, j), Array SpSk, Integer k)
16:    H_1 := h_1(T)
17:    T_1 := T sorted according to the values in H_1
18:    H_2 := h_2(T)
19:    T_2 := T sorted according to the values in H_2
20:    u_1 := T.length − 1
21:    u_2 := T.length − 1
22:    s := 0
23:    while s < T.length − 2 do
24:        while h_1(T_1[s]) + h_2(T_2[u_1 − 1]) > i do
25:            u_1 := u_1 − 1
26:        end while
27:        s_1 := u_1
28:        while h_1(T_1[s]) + h_2(T_2[u_2 − 1]) > i + k do
29:            u_2 := u_2 − 1
30:        end while
31:        s_2 := u_2
32:        while h_1(T_1[s]) + h_2(T_2[s_1]) < j do
33:            B := h_1(T_1[s]) + h_2(T_2[s_1]) mod k      ▷ compute the hash value
34:            SpaceSketch(SpSk[B], (T_1[s], T_2[s_1]))
35:            s_1 := s_1 + 1
36:        end while
37:        while h_1(T_1[s]) + h_2(T_2[s_2]) < j + k do
38:            B := h_1(T_1[s]) + h_2(T_2[s_2]) mod k      ▷ compute the hash value
39:            SpaceSketch(SpSk[B], (T_1[s], T_2[s_2]))
40:            s_2 := s_2 + 1
41:        end while
42:        s := s + 1
43:    end while
44: end procedure
```

Algorithm 5.1: *The initialisation on line 3 just put to 0 all the values in the entries of SpSk. An entry j in the array SpSk is:* $SpSk[j] = ((p_1, c_1, \epsilon_1), (p_2, c_2, \epsilon_2), c_j, s(p_1))$. *The* **While** *loop on Line 23 finds for every element x of a transaction T the items* $y \in T$ *such that* $h(x, y) \in [i, j) \cup [i, j) + k$ *(Loops on Lines 24 and 28). On Lines 34 and 39 are the calls to the procedure that maintains the sketches for both* SpaceSaving *and* Count-Sketch

```
 1: procedure SpaceSketch(Bucket SpSk[B], Pair p)
 2:     ((p1, c1, ε1), (p2, c2, ε2), cB) := SpSk[B]
 3:     if p = p1 then
 4:         c1 := c1 + 1
 5:     else
 6:         if p = p2 then
 7:             c2 := c2 + 1
 8:         else
 9:             p2 := p
10:             ε2 := c2
11:             c2 := c2 + 1
12:             cB := cB + s(p)                    ▷ s : P → {−1, 1}
13:         end if
14:         if c2 > c1 then
15:             swap((p1, c1, ε1), (p2, c2, ε2))
16:         end if
17:     end if
18: end procedure
```

Algorithm 5.2: *The pseudocode explains how the techniques* SpaceSaving *and* Count-Sketch *are implemented by the algorithm.*

requires more space for high quality estimates and our evaluations show that it performs rather poorly on real-life datasets compared to the estimates given by SpaceSaving.

5.4.1 SpaceSaving Based Sketch

Theorem 5.4.1. *For a Zipfian distribution with parameters C and $z > 1$, Algorithm 5.1 detects with constant error probability pairs with frequency at least $\Omega(\frac{C}{k^z})$; k is a user defined space usage parameter; that is, the hash function range. For $z < 1$ the bound is $\Omega(\max(\frac{C}{k^z}, \frac{Cd^{1-z}}{k}))$. We recall the reader that d is the number of distinct pairs occurring in all transactions. By allowing a multiplicative factor of $O(\log(\frac{1}{\delta}))$ for the space usage we report at least $(1-\delta)f$ frequent pairs with probability at least $1-\delta$ for a user-defined parameter δ.*

Proof. Let the minimum support threshold for frequent pairs be αm, $\alpha > 0$. (At the end of the proof we will obtain bounds on αm that depend on k.) We will estimate the probability that a frequent pair p is not reported. From the Zipf distribution we obtain that $x := (\frac{C}{\alpha m})^{\frac{1}{z}}$ pairs will have frequency above αm.

Let B be the bucket p hashes to. We first consider the case that another pair with frequency above αm will be hashed to B. This happens with probability $p_1 := x/k$. We will enforce a value for x, i.e. a lower bound on the support threshold of frequent pairs, such that $p_1 < 1/6$.

Assume now that the only frequent pair in B is p. As already discussed if the total weight of pairs frequencies hashed to B is less than $2\alpha m$, then p will be reported. Let $w := \sum_{i=x+1}^{d} C/i^z$ be the total weight of infrequent pairs. In the following we will use the fact that $w = O(C(d^{1-z}))$ for $z < 1$ and $w = O(C(x^{1-z}))$ for $z > 1$, where d is the number of distinct pairs in transaction

database. This follows by integration of the corresponding continuous function. Then we expect pairs of total weight w/k to land in B. In order to show a small deviation from the expected value we will adapt the analysis from [39] to our problem.

Let Y_j be an indicator random variable denoting whether the jth infrequent pair is hashed to B and $X = \sum_{j=1}^{d-x} Y_j$. Clearly, $\mathrm{E}[X] = w/k$. Applying Markov's inequality we obtain $\Pr[X \geq 3w/k] = p_2$. We will later enforce $p_2 \leq 1/3$. We want $3w/k \leq 2\alpha m$. For $z < 1$ we have $w \leq \frac{C(d^{1-z})}{1-z}$ thus we set $k = \max(6(\frac{C}{\alpha m})^{\frac{1}{z}}, 3\frac{C(d^{1-z})}{(1-z)(2\alpha m)})$. Similarly, for $z > 1$ we have $w \leq \frac{C(x^{1-z})}{(z-1)}$, hence the bound $k = \max(6(\frac{C}{\alpha m})^{\frac{1}{z}}, \frac{3C^{\frac{1}{z}}}{2(z-1)(\alpha m)^{\frac{1}{z}}})$.

Thus, for $z > 1$, we will consider pairs *frequent* if their frequency is $\alpha m = \Omega(\frac{C}{k^z})$ and for $z < 1$ if $\alpha m = \Omega(\max(\frac{C}{k^z}, \frac{Cd^{1-z}}{k}))$.

Thus, by the union bound the probability that a given pair with frequency at least αm is not reported is at most $p_1 + p_2 < 1/2$.

Instead of having one hash function we will work with $t := c \log \frac{1}{\delta^2}$, for some constant $c > 1$, independent hash functions such that each of them updates one of k unique SpaceSaving data structures. A pair will be reported as frequent only if it has won at least $t/2$ of its corresponding "races". Since the hash functions are independent and the expected number of a frequent pair being reported is at least $t/2$, we can apply Chernoff inequality and bound the probability of a frequent pair not reported to at most $O(\delta^2)$.

Note that the number of reported pairs is bounded by $2k$. Thus, we expect at most $\delta^2 x$ frequent pairs not to be reported and by Markov's inequality the probability we don't report more than δx frequent pairs is at most δ. □

We remark that the above analysis is for a worst case scenario, namely the one in which the pairs hashed to each bucket arrive in a specific order. Moreover, in order to derive theoretically sound analysis, we work with the pessimistic bounds given by Markov's inequality. In our experiments we show that for real-life datasets we are able to achieve very accurate results with much more modest parameters than the ones in the above analysis.

If we define the two functions:

Recall: the ratio between the number of identified frequent pairs divided by the number of all frequent pairs;

Precision: the ratio between the number of frequent pairs and the number of pairs that are output;

it is possible to balance between them by choosing an optimal value for the number of buckets k. The bigger k, the better recall we get, but the noise in the output decreases the precision. The two functions are good metrics for estimating the influence of the number of false positives and false negatives in the output.

5.4.2 Count-Sketch

Theorem 5.4.2. *For a Zipfian distribution with parameters C and $z > 1/2$ the unbiased estimator returned by Algorithm 5.1 estimates the frequency of a*

given pair with frequency $\Omega(\frac{(k/C^2)^{\frac{2z-1}{z}}}{C})$ *within an additive constant factor with a constant error probability. By allowing a multiplicative factor of* $O(\log(\frac{1}{\delta}))$ *for the space usage we report correct estimates for at least* $(1-\delta)f$ *frequent pairs with probability at least* $1-\delta$ *for a user-defined parameter* δ*.*

Proof. As in the original COUNT-SKETCH algorithm we work with an additional hash function: The sign function $s : \mathcal{P} \to \{-1, 1\}$.

In the analysis below we will obtain closed expressions for the frequency of items which we will consider frequent depending on C, z and the user-defined k. Let us first denote the minimum required support as $\sup(p) \geq \alpha m$, $\alpha > 0$. From the parameters of the Zipfian distribution we obtain a closed form for the number of pairs with frequency at least αm, i.e. $(\frac{C}{\alpha m})^{\frac{1}{z}}$, let us denote this number by x.

First, we have already analyzed the event that a given frequent pair is not reported. We will assume this happens with probability at most $1/2$.

We will now obtain upper and lower bounds on the estimates and will bound the probability of them being not incorrect. The expected value $\mathrm{E}[s(p) \cdot c_{h(p)}]$ is the exact frequency of p; abusing notation we denote by $c_{h(p)}$ the counter in the bucket where p hashes to. Our goal is to show concentration result around the expected value which will imply an high-quality estimate.

We bound the probability that pairs with total weight at least αm will land in $h(p)$. In order to show concentration around the expected value we will adapt the analysis from the previous section and will work with Chebyshev's inequality instead of Markov's. Let $w := \sum_{i=x+1}^{d} \frac{C}{i^z}$ denote the total weight of infrequent pairs. The sum of squared frequencies of infrequent pairs is $w_2 = \sum_{i=x+1}^{d} (\frac{C}{i^z})^2$. We use the fact that $w_2 = \frac{C^2 x^{1-2z}}{2z-1}$ for $z > 1/2$.

The expected excess in a bucket with a frequent pair, denoted by the random variable X, is $\mathrm{E}[X] = w/k$. The variance turns out to be $\mathrm{Var}[X] = w_2/k$. Thus, for $k > w_2$ we bound the probability for absolute deviation from expectation by more than λ to $1/\lambda^2$. We assume $\lambda > 2$, thus the error probability is at most $1/4$. Note that this bound directly implies an upper bound on the overestimation, respectively underestimation, of the frequency of the reported frequent pair. With some algebra one obtains the lower bound on the support claimed in the theorem.

The above lower bound on the space requirement is bigger than the bounds in Theorem 5.4.1t for $z > 1/2$.

With the same analysis of the previous section we compute the probability of not reporting the pair; therefore, to account also for the probability of over/under estimating the frequency, we can use a union bound and get an error probability smaller than $1/2$. The probability can be reduced to any desired δ by working with $t := c \log \frac{1}{\delta^2} = O(\log \frac{1}{\delta})$ independent hash (sign and bucket) functions for some constant $c > 1$, as outlined in the previous section. We will report a pair as frequent if and only if it has been found frequent at least $t/2$ times. We take the median of the results and apply Chernoff bounds in order to bound the probability the frequency has been over/underestimated. For bounding the number of not reported/not correctly estimated frequent pairs, we can apply again Markov's inequality as shown in the previous section. □

5.5 A lower bound

We present a lower bound for the space needed in order to report the most frequent pair in a stream of transactions. Of course, this applies also to any harder problem, such as reporting a larger set of frequent pairs, with counts. This lower bound complements the strong worst-case space lower bound presented in Chapter 4, by arguing that *any* data streaming algorithm for frequent pairs that does not make errors must store all information about itempair counts.

Theorem 5.5.1. *Any data streaming algorithm that always outputs the most frequent pair, or even just the frequency of the most frequent pair, must encode in its state all itempair counts, in the sense that if two stream prefixes differ in the count of some itempair, the algorithm must be in different states after seeing the prefixes.*

Intuitively, the only generally applicable ways of storing the counts of all pairs is to either store each count explicitly, or store all transactions seen so far.

Proof. For a prefix of the stream consider the count vector that for each pair records its frequency in the prefix. Let A be any algorithm that computes the most frequent pair in a data stream. Consider two distinct count vectors x and y, corresponding to different stream prefixes. We argue that A must be in two different states after seeing these prefixes. Suppose that the latter claim is not true, so that for $x \neq y$ the algorithm is in the same state. Since x and y differ, there must be at least a pair (u, v) having distinct counts in the two count vectors. But this implies that we can extend the streams with a sequence that makes the pair (u, v) the most frequent one in one of the vectors, say w.l.o.g. x, (u, v) becomes the most frequent pair, while this does not happen in y. Still, the algorithm would be in the same state in both cases, returning the same result and occurrence count. This contradicts the assumption that A always returns the correct answer, so the assumption that x and y resulted in the same state must be false. □

5.6 Experiments

Table 5.1 summarises the datasets that we use for experiments. In all cases, we use the order in which the transactions are given as the stream order. Experiments refer to the version of the algorithm using SPACESAVING. We worked with two implementations, a simple Python implementation, and a cache-optimised Java implementation that was 10–20 times faster. In both cases, we used the built-in random number generator of the language to store hash values in a table.

5.6.1 Pair similarity distribution

We would like to justify the assumption in our theoretical analysis that the counts over all pairs follow a Zipfian distribution. While it is well-known that this is true for single items in many datasets, it is not obvious that this assumption holds for streams of pairs generated from a stream of transactions. For this reason we computed the exact count the most significant pairs using

Dataset	# of pairs (F_2)	# of distinct pairs
Mushroom	$22.4 \cdot 10^5$	$3.65 \cdot 10^3$
Pumsb	$1360 \cdot 10^5$	$536 \cdot 10^3$
Pumsb_star	$638 \cdot 10^5$	$485 \cdot 10^3$
Kosarak	$3130 \cdot 10^5$	$33100 \cdot 10^3$
Retail	$80.7 \cdot 10^5$	$3600 \cdot 10^3$
Accidents	$187 \cdot 10^5$	$47.3 \cdot 10^3$
Webdocs	$2.0 \cdot 10^{11}$	$> 7 \cdot 10^{10}$
Nytimes	$1.0 \cdot 10^{10}$	$> 5 \cdot 10^8$
Pubmed	$1.6 \cdot 10^{10}$	$> 6 \cdot 10^8$
Wikipedia	$5.17 \cdot 10^{11}$	$> 5.8 \cdot 10^9$

Table 5.1: *Information on datasets for our experiments. Nytimes and Pubmed are taken from the UCI Machine Learning Repository (Bag of Words dataset). The Wikipedia dataset has been crafted according to what is described in [1, Page 14]. The rest are from the Frequent Itemset Mining Repository. For the last four datasets the number of distinct pairs was estimated using the hashing technique presented in Chapter 7. For Webdocs, only the first 100000 are taken into account.*

Borgelt's A-Priori implementation [1] [17], and plotted them in decreasing order. Figure 5.1 shows the supports of the most frequent pairs for our datasets. In all cases we see that the curve starts as approximately a decreasing straight line on a logarithmic plot. The length of this line varies from one dataset to another, from a few hundred pairs to hundreds of thousands. Observe also that in all cases where the curve deviates from a line it drops below, i.e., the distribution is dominated by a Zipfian distribution.

5.6.2 PairSE precision and recall

Our next set of experiments shows results on the precision of the counts obtained by PairSE, as well as the recall. Both aspects are of course influenced by the amount of space used. For each dataset we have chosen a relatively low space usage, to show that even with a small memory footprint good results are obtained. In practice, it may be hard to foresee how much space will be needed for a particular stream, so probably one will tend to use as much space as feasible with respect to running time (ensure in-cache hash table), or what amount of memory can be made available on the system. A consequence of this will be even more precise results. The results of the experiments are visible in Figures 5.2 and 5.3, where the former zooms in the zone where the lower and upper bound computed by SpaceSaving are very accurate (left part of Figure 5.3). The influence of the space usage on the recall, can also be appreciated looking at Figure 5.4; the relative experiment is described in Section 5.6.3.

5.6.3 PairSE space requirements

We now investigate what happens to precision and recall when the space usage of PairSE is pushed to, and beyond, its limits. In order to do this, we chose to

[1] `http://www.borgelt.net/apriori.html`

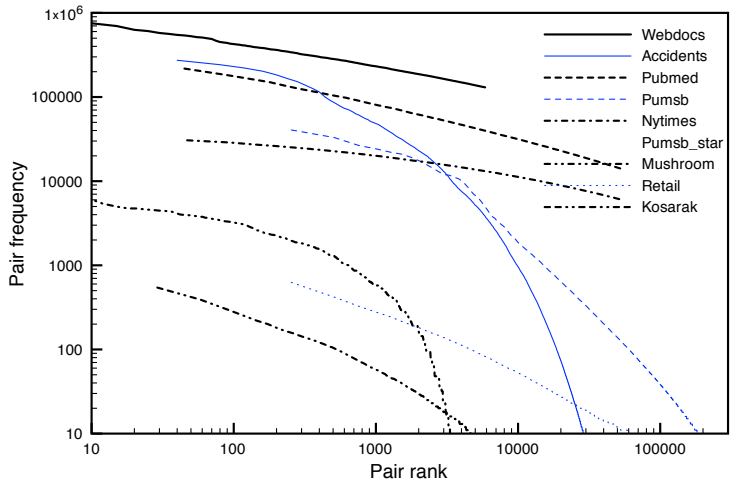


Figure 5.1: *The frequency distribution for the most frequent pairs, on doubly-logarithmic scale. All start off with a straight line.*

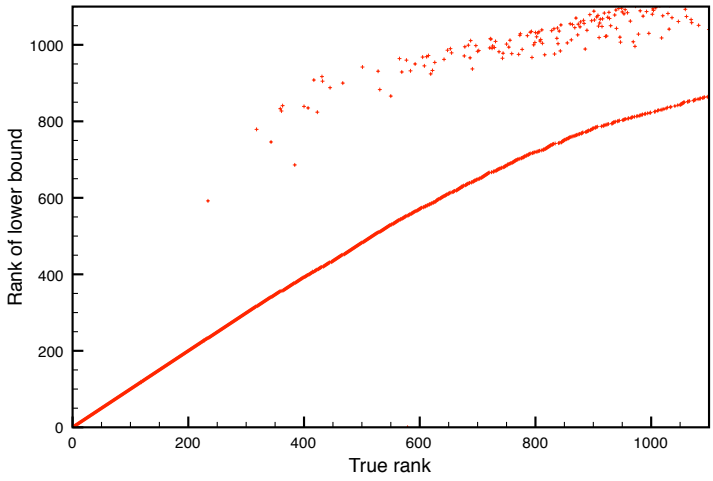


Figure 5.2: *Top frequent pairs for Webdocs, and their rank according to the frequency lower bound computed by PairSE using* 2^{20} *buckets. As can be seen, recall is initially high, but decreases with the support.*

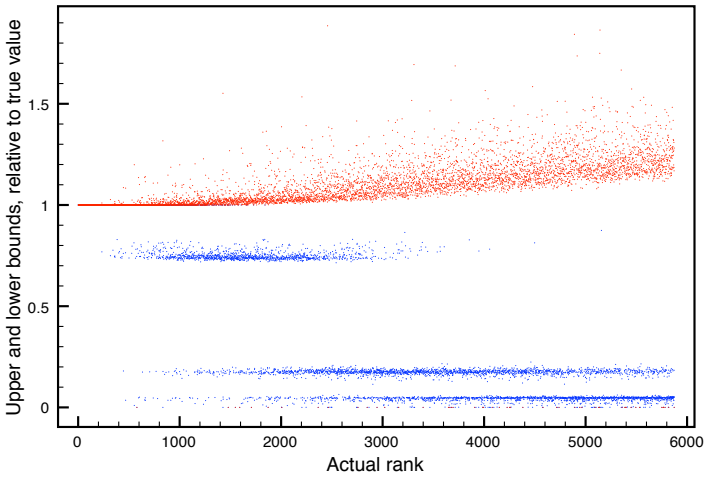


Figure 5.3: *Upper and lower bounds for Webdocs computed by PairSE using* 2^{20} *buckets. Values are normalised by dividing by true support. Upper bounds shadow lower bounds; exact bounds, so matching upper and lower bounds, are visible only as red dots without blue dots below, and are obtained for the most frequent pairs. As can be seen, upper bounds are generally tighter than lower bounds.*

work with 3 representative datasets, namely Mushroom, Retail and Accidents. We decreased the space usage gradually, plotting the ratio between the upper and lower bounds for the top-100 pairs returned by our algorithm. This is shown in Figure 5.4 and we can see how the transition between very good and very poor quality is fairly fast.

5.6.4 Load balance

Balancing of pairs We ran experiments in order to evaluate the distribution of pairs amongst the buckets. When running these experiments, we kept track also of the number of pairs that were evicted by a bucket, the number of swaps, the number of increments to the counters. The results of these experiments are reported in Table 5.2. The numbers in the table confirm that the pairs spread evenly amongst the buckets; this means that parallelism greatly improves the running time of the algorithm, since there will be no core that have to sustain a much larger burden than the others; such a negative situation would bring the performances of the algorithm close to a sequential one.

Progress of processing To illustrate the load balancing in terms of running time, we ran PairSE on 4 cores of an Intel Core i7 Q720 1.60GHz laptop running GNU/Linux, kernel version 2.6.38.7 vanilla. Parallelism was achieved by running four processes, and letting the operating system allocate one to each core. We tracked the progress of each process over time. The results for two representative data sets can be seen in figures 5.5 and 5.6. As can be seen, the cores make almost identical progress when running at full speed. In a data streaming setting this means that we can expect to manage streams that require all cores to run

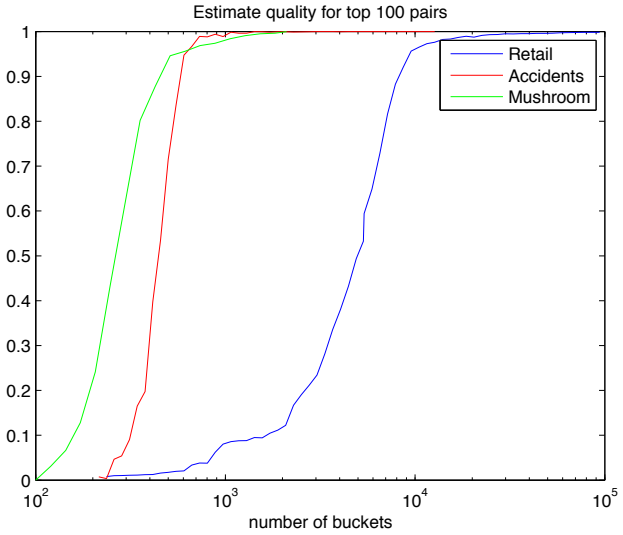


Figure 5.4: *Average ratio of lower and upper bound for top-100 pairs, for three representative datasets. On the x axis we have the number of buckets. On the y axis the ratio mentioned before. As can be seen, there is a quick transition from poor to excellent precision.*

Dataset	Cores	Average	Maximum
Retail	8	895542	934040
	4	1791084	1808784
	2	3582168	3585869
Kos	8	3203546	3219769
	4	6407091	6427398
	2	$1.2814 \cdot 10^7$	$1.2827 \cdot 10^7$
Webdocs	8	$8.928 \cdot 10^8$	$8.9394 \cdot 10^8$
	4	$1.7856 \cdot 10^9$	$1.7871 \cdot 10^9$
	2	$3.5712 \cdot 10^9$	$3.573 \cdot 10^9$
Nytimes	8	$1.2497 \cdot 10^9$	$1.251 \cdot 10^9$
	4	$2.4995 \cdot 10^9$	$2.5005 \cdot 10^9$
	2	$4.9989 \cdot 10^9$	$4.9995 \cdot 10^9$
Wikipedia	8	$6.4582 \cdot 10^{10}$	$6.4651 \cdot 10^{10}$

Table 5.2: *The table shows the average and maximum number of pair occurrences handled by each core. As can be seen, the maximum is quite close to the average.*

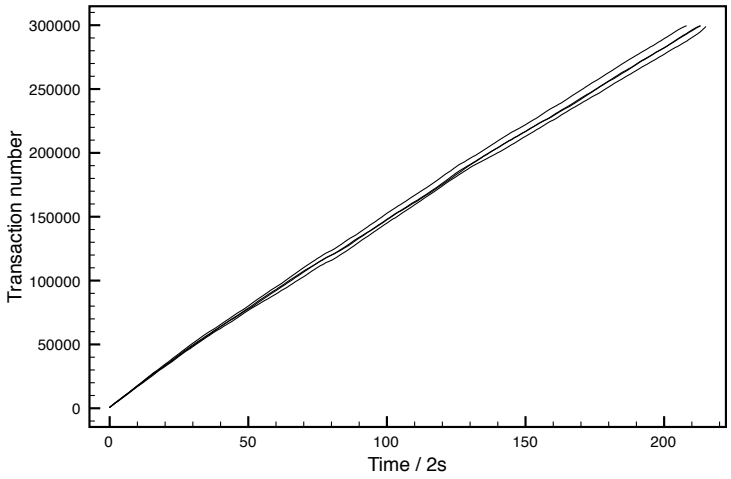


Figure 5.5: *Current transaction number of each core running PairSE on the Nytimes data set, over a period of about 400 seconds.*

at close to maximum speed, while needing to cache only a small number of transactions.

5.6.5 Performance and scalability

Experiments have been carried out in order to verify how the algorithm scales, in terms of time, when parallel computations are used. We ran the algorithm on various datasets using several different number of cores. In this way it has been possible to highlight the parallel nature of the algorithm, hence, its capability of being very time efficient when many cores are at hand. Table 5.3 reports some of the results we obtained. The machine we used is described in the caption of the table. It is interesting to point out that in case of large datasets, with a high number of pairs, the algorithm allowed the CPU to manage almost 100 millions pairs per second. This means that the algorithm scales particularly well when the transaction are large. In those cases the parallelism of the computation speeds up considerably the algorithm.

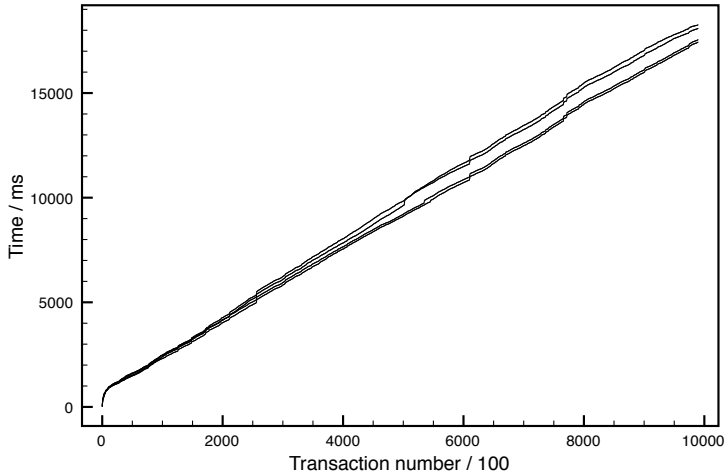


Figure 5.6: *Total time recorded for every* 100*th transaction when running PairSE with 4 cores on the Kosarak data set.*

Dataset	**# of cores**	***ms* on # cores**	***ms* 1 core**
Retail	8	1234	1897
	4	1050	
	2	1342	
Docword.kos	8	822	2165
	4	824	
	2	1260	
Webdocs	8	72687	711685
	4	137308	
	2	262075	
Nytimes	8	132754	1120705
	4	223724	
	2	435922	
Wikipedia	8	11326791	

Table 5.3: *Experiments ran on an Intel Xeon E5570 2.93 Ghz equipped with 23 GB of RAM; the OS is GNU/Linux, kernel version 2.6.18. The number of processes used is 8 for all the four rows. Times are given in milliseconds (ms). We can observe that in any case, millions of pairs per second were manipulated by the algorithm.*

Chapter 6

On Finding Frequent Patterns in Event Sequences

Given a directed acyclic graph with non unique labels associated to vertices, we consider the problem of finding the most common label sequences ("traces") among all paths in the graph (of some maximum length m). Since the number of paths can be huge, we propose novel algorithms whose time complexity depends only on the size of the graph, and on the frequency ε of the most frequent traces. In addition, we apply techniques from streaming algorithms to achieve space usage that depends only on ε, and not on the number of distinct traces.

The abstract problem considered models a variety of tasks concerning finding frequent patterns in event sequences. Our motivation comes from working with a dataset of 2 million RFID readings from baggage trolleys at Copenhagen Airport. The question of finding frequent passenger movement patterns is mapped to the above problem. We report on experimental findings for this dataset.

6.1 Introduction

Sequential pattern mining has attracted a lot of interest in recent years. However, some of the probabilistic techniques that have proved their efficiency in mining of frequent itemsets have, to our best knowledge, not been transferred to the realm of sequence mining. In this chapter we take a step in that direction, namely, we propose an analogue of Toivonen's sampling-based algorithm for frequent itemset mining [90] in the context of sequential patterns.

At a conceptual level we work with a new, simple formulation of the problem: The input is a directed acyclic graph (DAG) where the vertices are events and there is an edge between two events if they are considered to be connected (i.e., part of the same event sequences). Vertices are labelled by the type of event they represent. This allows certain flexibility in modelling that is lacking in many other formulations:

- Spatio-temporal events can be connected based on both spatial and temporal closeness.

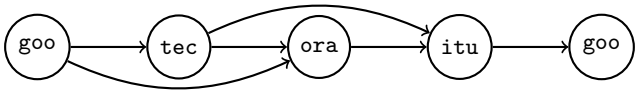


Figure 6.1: *Example of a browsing history represented by a DAG*
The browsing history of a user session can be captured by the dag, which also accounts for detours or visits to search engines.

- Events that have an associated time range (rather than a single time stamp) can be connected based on an arbitrary closeness criterion.

The data mining task we consider is to find the most common sequences of event types ("traces") among all paths in the DAG, or more generally all paths of some maximum length m. The challenge is to handle the huge number of paths that may be present in a DAG.

Example 6.1.1. Consider data on the history of URLs visited by a user, where each URL is labelled by its domain name. If the user visits the domains `www.techcrunch.com`, `www.oracle.com`, and `www.itu.dk` in this order, there may be a connection between the first and second site, and between the second and third site. If all visits happen within a few minutes one could also imagine that the second site was merely a detour, and there is a connection from the first to the third site. This is naturally modelled using a graph having URL visits as vertices, and directed edges between vertices that we deem connected (based on any criterion, e.g., temporal closeness). We label vertices by domain name, and look for frequently occurring label sequences, *traces*, on paths in the graph. Figure 6.1 gives represents the situation that we have just described. ◦

We might be interested in such frequent event sequences for a variety of reasons, e.g. improved understanding of browsing behavior for advertisers (avoid paying for many page impressions to the same user), and page recommendations ("users who visited the same sequence of domains as you, often went on to the domain. . . "). We should be able to detect the connection between sites even if they are not visited in succession. For example, many browsing histories will interleave visits to hubs such as `google.com` and `yahoo.com` with visits to topic specialized domains.

6.1.1 Approach

We start from the observation that the number of paths in a DAG can be extremely large, even if the path length is restricted to some small number m. For example, the DAG pictured in Figure 6.2 has 16 vertices and 45 edges, but the number of paths is 10919.

More generally, we expect the number of paths to increase exponentially with m. In our experiments we see that, even for small m, the number of paths is much larger than the size of the DAG.

Our algorithm rests on a novel *sampling* procedure that is able to create a sample of any desired size, in time that is linear in the size of the DAG (for

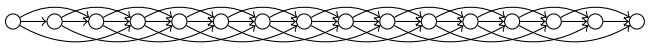


Figure 6.2: *In this small DAG, there are 16 vertices, 45 edges but the number of paths is 10919. The number of paths can have an exponential dependence on the maximum length of a path.*

preprocessing) and the size of the sample. This allows a time complexity for the mining procedure that depends only on the *frequency* ε of the most common traces, rather than the total number of traces. We also apply a technique from data streaming algorithms to achieve space that depends on ε rather than on the number of distinct traces.

Though our formulation does not capture all the many aspects present in other approaches to sequential pattern mining, we believe that it possesses an attractive combination of *expressive modeling* and *algorithmic tractability*.

6.1.2 Problem definition

We are given a directed acyclic graph $G = (V, E)$, and a function label(v) that returns the label of a vertex. A path p in G is a sequence of vertices $v_1, v_2, \ldots, v_j \in V$ such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, j-1$. A path p has a *trace* label(p), which is the vector of labels on the path. Let S_m be the multiset of all path traces of length at most m, i.e.,

$$S_m = \{\text{label}(p) \mid p \text{ is a path in } G \text{ of length at most } m\} \ .$$

The data mining task is to find the most frequent traces in S_m. It comes in several flavours:

Top-k: For a parameter k, find the k traces that have the most occurrences in S_m (breaking ties arbitrarily).

Frequency ε: Find the set of traces that have frequency ε or more in S_m.

Monte Carlo: For both the above variants we can allow an error probability δ (typically allowing a false negative probability, i.e., that we fail to report a trace with probability δ).

In this chapter, emphasis will be on Monte Carlo algorithms for the frequency variant. However, we note that one can also obtain results for top-k by a simple reduction.

6.1.3 Related work

There is a large body of related work on sequential pattern mining, see e.g. [77, 89, 67, 61, 99, 51, 31, 87]. These works deviate from the present one in that they consider the input as a sequence of timestamped events, and allow a host of formulations of what kinds of subsequences are of interest. In contrast, we put the modeling of interesting subsequences into the description of the event sequence (by defining DAG edges), and the patterns sought are simple strings. This allows us to do things that we believe have not been done, and are probably

difficult, in traditional sequential data mining settings, namely making use of sampling methods.

The difficulty with sampling is that patterns can overlap in many ways, so any straightforward approach will fail to produce a sample that correctly "represents" the original data. As an example, suppose that the pattern $\mathtt{a}^{2m}$ occurs in the input, which means $k+1$ occurrences of $\mathtt{a}^m$. If we sample events with probability 50%, the probability that an occurrence of $\mathtt{a}^m$ remains in the sample is $1/2$. On the other hand, if there are $k+1$ non-overlapping occurrences of $\mathtt{a}^m$, the probability that this is seen in the sample may be much lower. For example, for the string $(\mathtt{a}^m\mathtt{b}^m)^{m+1}$ the probability is $O(m/2^m)$, i.e., exponentially decreasing as m grows. This means that there is no direct way of going from the number of occurrences in the sample to the number of occurrences in the original string.

Similar problems make use of sampling methods in general graph mining difficult. Suppose that we sample vertices (or edges) with probability p. If all triangles in a graph overlap in a single vertex, the sample will contain no triangles at all with probability $1-p$. On the other hand, if there is the same number of vertex (edge) disjoint triangles, we are likely to sample close to a fraction p^3 of them. As before, we cannot estimate the number of occurrences in the original graph based on the number of occurrences in the sample.

6.2 Our solution

6.2.1 Generation of all traces

As a warmup we consider the task of producing the multiset S_m of all traces having maximum length m. We will use the notation $S_i(v)$ to denote the multiset of traces corresponding to paths (of length at most i) starting in node v. Clearly $S_0(v) = \emptyset$. For $i > 0$ we have the recursive definition

$$S_i(v) = \{\text{label}(v)\} \times (\epsilon \cup \bigcup_{v',\,(v,v')\in E} S_{i-1}(v')),$$

where ϵ denotes the empty trace (note that this symbol is different from ε denoting the frequency), and $\bigcup$ is multiset union. Clearly we have $S_m = \bigcup_{v\in V} S_m(v)$.

These equalities lead to a simple recursive algorithm, shown in Algorithm 6.1. It is easy to see that if traces are represented in a reasonable way (e.g. as singly linked lists) the running time is linear in the size $|V|+|E|$ of the graph and the total length of the traces generated.

Succinct output. If we are satisfied with returning hash values of the traces (unique with high probability) the time can be improved such that only $O(1)$ time is used for each trace, i.e. time $O(|V| + |E| + |S_m|)$ in total. This can be done using a standard incremental string hashing method such as the one that can be found in [69]. Observe that the output is sufficient to find the *hash values* of the most frequent traces in S_m (with a negligible error probability). A second run of the procedure could then output the actual frequent traces, e.g. by looking up the count of each hash value computed.

```
 1: procedure AllTraces(v, t, i)
 2:     if i > 0 then
 3:         output t||label(v)
 4:         for each v' where (v, v') ∈ E do
 5:             AllTraces(v', t||label(v), i − 1)
 6:         end for
 7:     end if
 8: end procedure

 9: for v ∈ V do
10:     AllTraces(v, ϵ, m)
11: end for
```

Algorithm 6.1: *The procedure* AllTraces *outputs the concatenation of a trace prefix t, and each trace starting at v having length at most i. The notation $||$ is for concatenation of traces. Lines 7–9 call* AllTraces *for all vertices v, with the empty trace ϵ as prefix, producing the multiset S_m of all traces of length at most m.*

6.2.2 Generation of a random sample

If the patterns we are interested in occur many times, substantial savings in time can be obtained by employing a sampling procedure. That is, rather than generating S_m explicitly we are interested in an algorithm that produces each trace in S_m with a given probability p, independently. This will reduce the expected number of samples to a fraction p of the original. The choice of p is constrained by the fact that we still want to sample each frequent trace a fair number of times (to minimize the probability of *false negatives* being introduced by the sampling).

Counting phase. Our algorithm starts by computing, for $i = 1, \ldots, m$ the number of paths $v.c[i]$ of length at most i that start in each vertex v. We assume that this can be done using standard precision (e.g. 64 bit) integers. The pseudocode shown in Algorithm 6.2 mimics the structure of the naïve generation algorithm, but uses memoization (aka. dynamic programming) to reduce the running time.

For each $i \leq m$ the cost of all calls to CountTraces with parameters (v, i), disregarding the cost of recursive calls, is easily seen to be proportional to the number of edges incident to v. This means that the total time complexity of the counting phase is $O(|E|m)$. The space usage is dominated by an array of size m for each vertex, i.e., it is $O(|V|m)$.

Sampling phase. Consider the multiset $S_i(v)$ of traces, which has size $v.c[i]$ by definition. The probability that none of these traces are sampled should be $(1-p)^{v.c[i]}$. Conditioned on the event that at least one trace from $S_i(v)$ is sampled, we either have to sample a trace of length more than one (starting with label(v)), or include the trace $\{v\}$ in the sample. In a nutshell, this is what the procedure SampleTraces of Algorithm 6.3 does.

Let rand() denote a function that returns a uniformly random number in $[0; 1]$, independently of previously returned values. The condition rand() $>$

```
 1: function CountTraces(v, i)
 2:     if v.c[i] =null then
 3:         v.c[i] := 1
 4:         for each v' where (v, v') ∈ E do
 5:             v.c[i] := v.c[i]+CountTraces(v', i − 1)
 6:         end for
 7:     end if
 8:     return v.c[i]
 9: end function
10: for v ∈ V do
11:     CountTraces(v)
12: end for
```

Algorithm 6.2: *Recursive computation of the paths of traces for each starting vertex, using memoization. The algorithm assumes that each value $v.c[0]$ is initially set to zero, and each value $v.c[i]$, $0 < i \leq m$, is initially* **null**.

$(1-p)^{v.c[m]}$ holds with probability $1-(1-p)^{v.c[m]}$, so lines 14–16 call SampleTraces if and only if we need to sample at least one trace from $S_m(v)$. In the procedure SampleTraces we use, similarly to above, a parameter t to pass along a trace prefix. The variable *out* is used to keep track of whether a trace has been output in the recursive calls. If *out* is false after all recursive calls we sample $t||\text{label}(v)$. For each v' with $(v,v') \in E$ the probability that we do *not* sample any trace from $\text{label}(v)||S_{i-1}(v')$ is $(1-p)^{v'.c[i-1]}/(1-(1-p)^{v.c[i]})$. This is exactly the correct probability since we condition on at least one trace in $S_i(v)$ being sampled.

Refinement. Looking at the pseudocode of Algorithm 6.3, we can observe that the probability in Line 4 may be precomputed for each edge and value of i. Even with this optimization, a direct implementation of that pseudocode may spend a lot of time in the **for** loop of 3 without producing any output. To get a theoretically satisfying solution we may preprocess, for each (v,i), the probabilities $p_1, p_2, \ldots, p_d$ of making the recursive calls. Specifically, for $j = 0, \ldots, d$ we consider the probabilities $q_j = \Pi_{j' \leq j}(1-p_{j'})$ that no recursive call is made in the first j iterations. If we choose r uniformly at random in $[0;1]$ then the probability that $q_{j-1} > r > q_j$ is exactly the probability that the first recursive call is in the jth iteration. Similarly, the probability that $r < q_d$ is exactly the probability that no recursive call is made. Thus, by doing a binary search for r over $q_d, \ldots, q_0$ we may choose, with the correct probability, the first iteration j_1 in which there should be a recursive call. The same method can be repeated, using a random value r in $[0; q_{j_1}]$ to find the next recursive call, and so on.

In the worst case this uses time $O(\log |V|)$ per recursive call. We can exploit the fact that we are searching for a random value r to decrease this to $O(1)$ expected time. The idea is to represent the values q_j in a binary trie that is precomputed for each node. In addition we store for each string $s \in \{0,1\}^{\lceil \log d \rceil}$ a pointer to the node in the trie that corresponds to the longest prefix of s. The number of bits of r needed to determine its position in $q_d, \ldots, q_0$ is at most $\lceil \log d \rceil + t$ with probability at least $1 - 2^{-t}$. Using the pointers we can thus in

```
1: procedure SampleTraces(v, t, i)
2:     out := false
3:     for each v' where (v, v') ∈ E do
4:         if rand()> (1 − p)^{v'.c[i−1]}/(1 − (1 − p)^{v.c[i]}) then
5:             SampleTraces(v', t||label(v), i − 1)
6:             out := true
7:         end if
8:     end for
9:     if out = false or rand()< p then
10:        output t||label(v)
11:    end if
12: end procedure

13: for v ∈ V do
14:    if rand()> (1 − p)^{v.c[m]} then
15:        SampleTraces(v, ϵ, m)
16:    end if
17: end for
```

Algorithm 6.3: *The procedure* SampleTraces *outputs the concatenation of a trace prefix t and a random sample of the traces starting at v of length at most i. The traces are sampled from the conditional distribution that is guaranteed to sample at least one trace. As before, the notation $||$ is for concatenation of traces, and ϵ denotes the empty trace. Lines 13–17 call* SampleTraces *for each vertex v with probability $1 - (1-p)^{v.c[i]}$, to produce a sample of all traces starting at v having length at most i, where each trace is chosen independently at random with probability p.*

expected time $O(1)$ find the node in the trie that has the longest common prefix with the binary representation of r. This, in turn, determines the rank of r in $q_d, \ldots, q_0$.

As before, we can choose to have a succinct output where traces are represented by the hash values of their traces, with no increase in time complexity.

6.2.3 Time and error analysis

For the time analysis we focus on the refined implementation described above, since it allows a clean and exact theoretical analysis. A similar analysis of the version stated in the pseudocode can be made under the assumption that the outdegree of vertices in G is bounded by a constant. Observe that if SampleTraces makes c recursive calls this takes expected time $O(1+c)$. Also observe that the total number of procedure calls is upper bounded by the total length of all sampled traces — this is because each recursive call is guaranteed to produce at least one output. Combining these facts we see that the expected time for all calls to SampleTraces is linear in the length ℓ of all traces sampled. Notice that the expected value of ℓ is $O(p|S_m|m)$. Since ℓ is independent of the random choices determining the running time of the data structure in the refined implementation we can conclude that the total expected running time of the code in Algorithms 6.2 and 6.3 is $O(|V| + |E|m + p|S_m|m)$.

The parameter p must be chosen such that $p = C/\varepsilon$, where $C > 1$ is a parameter that determines the false negative probability. The expected number of times that we sample a trace with frequency ε' is $C\varepsilon'/\varepsilon$, and since the samples are independent, the number of samples follows a binomial distribution. By Chernoff bounds, this means that if $\varepsilon' \geq \varepsilon$ then the number of samples is at least $C/2$ with probability $1 - 2^{-\Omega(C)}$. Examples of concrete error probabilities are given in our experimental section. We have the following theoretical result:

Theorem 6.2.1. *We can generate a random sample of S_m in expected time $O(|V| + |E|m + \log(1/\delta)/\varepsilon)$ such that any trace with frequency ε or more has frequency at least $\varepsilon/2$ in the random sample with probability $1 - \delta$.* ∘

Observe that the running time is independent of the total number of traces in S_m.

6.2.4 Putting things together

It remains to assess how to choose, among the samples, the ones that are actually interesting. In particular, we are interested in those traces appearing in the sample at least $C/2$ times.

This problem can be efficiently faced using a *frequent items* algorithm. Such algorithms are widely used in data streaming contexts, and guarantee very small space usage. A comprehensive treatment and an experimental comparison between various techniques can be found in [37].

Definition 6.2.1. *Given a stream $\mathcal{S}$ of n elements, a frequency threshold η, and let f_i be the the frequency of i in $\mathcal{S}$. The* frequent items *problem consists in returning a set $\mathcal{F}$ of size at most $1/\eta$ such that for all i with $f_i > \eta$, $i \in \mathcal{F}$.*∘

Observe that false positives, with $f_i < \eta$, can appear in the output. To eliminate these, we simply make another pass (i.e., generate the same sample again) to compute exact frequencies.

Theorem 6.2.2. *Given a stream of elements representing the set of samples of traces produced by* SampleTraces*, the space needed in order to output the traces with frequency at least $\varepsilon/2$, without producing any trace with frequency less than $\varepsilon/2$, is $O(1/\varepsilon)$ words.* ∘

6.3 From event sequence to a DAG

An event sequence is a set S of tuples of the form (t, i, ℓ), where $t \in \mathbb{R}$ is a time stamp, i is a tag identifier, and ℓ is a label (in our application case of RFID readings from baggage trolleys, i identifies the RFID on a trolley and ℓ is a location identifier that indicates an approximate location, namely vicinity of an antenna, of i at time t). In this work we do not consider the physical locations of antenna as part of the input.

Formally we may define the problem as follows: For a given number Δ, the input set specifies a directed acyclic graph $G_\Delta = (V, E_\Delta)$, where each observation is a vertex, and there is an edge from v_1 to v_2 if and only if the vertices are observations of the same tag, at different locations, separated by at most Δ time units (we use minutes as the time unit from now on).

To produce the DAG we sort the data by tag ID and timestamp. Note that this makes it easy to find all the edges from a particular vertex v in G_Δ: Simply scan the sorted list forward until either the timestamp differs by more than Δ from that of v, or we reach a node corresponding to another tag.

Example 6.3.1. If $\Delta = 20$ and we observe locations 1, 2, 3, 6, 7 at time 10, 20, 30, 60, 70, the following subsequences are considered to reflect a movement: 1-2, 2-3, 1-2-3, 1-3, 6-7. Notice the inclusion of 1-3, where one observation is skipped, since there is at most Δ minutes between the observation of 1 and 3. ∘

6.4 Experiments

We have worked with a dataset consisting of readings of RFID (Radio-Frequency ID) tags by fixed-position antenna. RFID chips can be identified only when they are in the proximity of an antenna, which means that readings give approximate information about the location of an RFID tag. Such datasets, as well as similar datasets based on other technologies, are becoming increasingly available as more and more items, from parcels to items in shops, are being tagged with RFID chips.

In order to construct the DAG, we have cleaned some of the noise present in the data. One source of noise was due to the presence of sequences of readings regarding trolleys remaining in zones where the range of two antennas is overlapping. This sequences of alternating readings had the form $(x^+y^+)(x^+y^+)^+$. In order to clean up this interferences, we replaced the elements of such a kind of sequences, using a new zone label that represents the zone of overlap of the

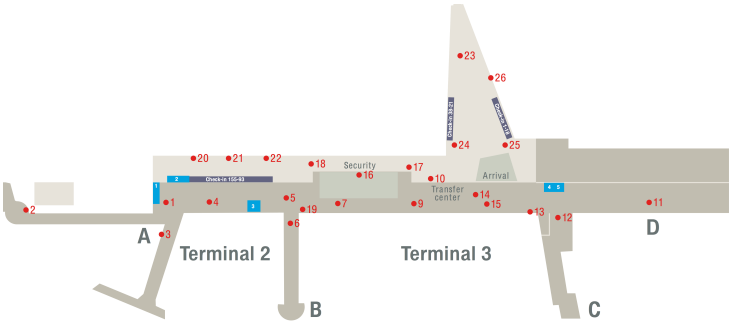


Figure 6.3: *RFID antenna in Copenhagen Airport.*

range of antennas. In particular we have used, for a sequence $(x^+y^+)(x^+y^+)^+$, the label $\min\{x, y\} * 100 + \max\{x, y\}$.

Notice that this can be thought as an increase in the resolution of the readings, making the granularity of the information finer. In some sense this modification allows for a cleaner sight on the movement of some trolleys.

Another source of noise, sometimes connected with the one just described, is the presence of sequences of readings regarding the same zone for a given trolley. In order to avoid having traces of the form $t = (Qyy^+W)$, where Q and W are sequences of readings, we considered only one occurrence of y, properly managing the timestamps of the readings. In particular this means that, assuming the difference in time between any two consecutive y is within the threshold Δ, in the DAG we put a directed edge (v, y), $v \in Q$ iff the first occurrence of y after Q occurred within time Δ from v. Moreover we put a directed edge (y, w), $w \in W$ iff w happened within time Δ from the last reading of y in t.

It is necessary to point out that our method differs from the previous approaches in the way we look for frequent patterns. This means that our results are not directly comparable with the ones that can be found in literature, so we do not compare to existing algorithms.

6.4.1 Results

We ran a set of experiments on the airport data, in order to understand how many patterns would have been generated for a given Δ and a size m. Table 6.2 shows the size of the graph for different sizes of Δ. We compare the obtained results with the expected performance of our algorithm.

Table 6.2 reports some interesting characteristics of the data when fixing Δ and m. In particular the table contains the number of traces generated, the frequency of the 100th most frequent trace and the ratio between the space needed in case of an exact computation and the space required when our algorithm is used. Note that the space to represent the DAG and the counts is not counted in this ratio. The rationale for this is that as we consider longer event sequences the space for the DAG representation is expected to become negligible compared to the space needed for finding the most common traces.

From the results of the test it is clear that great savings can be achieved

Δ	$\lvert V\rvert$	$\lvert E\rvert$
20	2206302	4059250
10	2206302	2657931
5	2206302	1721448
3	2206302	1228759

Table 6.1: *Size of the airport DAG for different values of Δ. As can be seen all graphs are quite sparse, and in fact many nodes have no outgoing edges. This is due to a relatively low resolution in the dataset.*

Δ	m	Tot. traces	Dis. traces	top 100th	ratio
20	5	365818472	4311942	168000	990
10	5	106678064	1712646	52951	425
10	3	6196850	50085	9458	38.2
5	5	66947355	631300	42008	198
3	5	23152990	280454	15363	93

Table 6.2: *Characteristics of the data for several combinations of Δ and m. The third column, Tot. traces, represents the total number of traces that would be generated by the naïve approach; the Dis. traces column represents the number of distinc traces; the top $100th$ column contains the frequency of the $100th$ most frequent trace; the column ratio represents the saving we would achive using a frequency threshold equal to the one represented in the top $100th$ column.*

when the frequencies we are interested in are not too low. In a case, nearly 3 orders of magnitude of space can be saved using our approach. As a matter of fact, when we are interested in very frequent traces, and this is often the case in many practical applications, the sampling outputs a large number of samples for each interesting trace, so that a low sampling ratio can be used.

Table 6.3 shows the number of samples we would take in expectation when $C = 10$ is used. The table gives the flavor of the saving in time that could be achieved with respect to generating all the possible traces. Here we notice that the total number of traces is already 1–2 orders of magnitude larger than the size of the DAG, so we expect an improvement in running time of at least 1 order of magnitude. Larger values of C will increase the running time proportionally, but decrease the error probabilities. Table 6.4 shows false negative probabilities, as well as probabilities that traces with frequency below $\varepsilon/4$ are reported.

Δ	m	Tot. traces	# samples	ratio
20	5	365818472	22774	16800
10	5	106678064	20147	5295
10	3	6196850	6552	946
5	5	66947355	15937	4200
3	5	23152990	15070	1536

Table 6.3: *The ratio between the total number of traces and the number of samples we would take using $C = 10$.*

C	False negative probability	Significantly false positive probability
3	0.199	0.173
5	0.125	0.127
10	0.0671	0.0420
15	0.0180	0.0376
20	0.0108	0.0318
30	0.00195	0.0103

Table 6.4: *Probability that a trace with frequency ε or more is not reported (false negative), and probability that a trace with frequency less than $\varepsilon/4$ is reported (significantly false positive), for different values of parameter C. The values are computed using the Poisson approximation to the binomial distribution, which is accurate unless the set S_m from which we sample is small.*

Chapter 7

Size Estimation for Sparse Matrix Products

We consider the problem of doing fast and reliable estimation of the number z of non-zero entries in a sparse boolean matrix product. This problem has applications in databases and computer algebra.

Let n denote the total number of non-zero entries in the input matrices. We show how to compute a $1 \pm \varepsilon$ approximation of z (with small probability of error) in expected time $O(n)$ for any $\varepsilon > 4/\sqrt[4]{z}$. The previously best estimation algorithm, uses time $O(n/\varepsilon^2)$. We also present a variant using $O(\text{sort}(n))$ I/Os in expectation in the cache-oblivious model.

In contrast to these results, the currently best algorithms for computing a sparse boolean matrix product use time $\omega(n^{4/3})$ (resp. $\omega(n^{4/3}/B)$ I/Os), even if the result matrix has only $z = O(n)$ nonzero entries.

Our algorithm combines the size estimation technique of [15] with a particular class of pairwise independent hash functions that allows the sketch of a set of the form $\mathcal{A} \times \mathcal{C}$ to be computed in expected time $O(|\mathcal{A}| + |\mathcal{C}|)$ and $O(\text{sort}(|\mathcal{A}| + |\mathcal{C}|))$ I/Os.

We then describe how sampling can be used to maintain (independent) sketches of matrices that allow estimation to be performed in time $o(n)$ if z is sufficiently large. This gives a simpler alternative to the sketching technique found in [47], and matches a space lower bound shown in that paper.

Finally, we present experiments on real-world datasets that show the accuracy of both our methods to be significantly better than the worst-case analysis predicts.

7.1 Introduction

In this chapter we will consider a $d \times d$ boolean matrix as the subset of $[d] \times [d]$ corresponding to the nonzero entries. The product of two matrices R_1 and R_2 contains (i, k) if and only if there exists j such that $(i, j) \in R_1$ and $(j, k) \in R_2$. The matrix product can also be expressed using basic operators of relational algebra: $R_1 \bowtie R_2$ denotes the set of tuples (i, j, k) where $(i, j) \in R_1$ and $(j, k) \in R_2$, and the projection operator π can be used to compute the tuples (i, k) where there exists a tuple of the form $(i, \cdot, k)$ in $R_1 \bowtie R_2$. Since most of

our applications are in database systems we will primarily use the notation of relational algebra.

We consider the following question: Given relations R_1 and R_2 with schemata (a, b) and (b, c), estimate the number z of *distinct* tuples in the relation $Z = \pi_{ac}(R_1 \bowtie R_2)$. This problem has been referred to in the literature as *join-project* or *join-distinct*[1]. We define $n_1 = |R_1|$, $n_2 = |R_2|$, and $n = n_1 + n_2$. As observed above, the join-project problem is equivalent to the problem of estimating the number of non-zero entries in the product of two boolean matrices, having n_1 and n_2 non-zero entries, respectively.

In recent years there have been several papers presenting new algorithms for sparse matrix multiplication [9, 75, 96]. In particular, these algorithms can be used to implement boolean matrix multiplication. However, the proposed algorithms all have substantially superlinear time complexity in the input size n: On worst-case inputs they require time $\omega(n^{4/3})$, even when $z = O(n)$.

In an influential work, Cohen [32] presented an estimation algorithm that, for any constant error probability $\delta > 0$, and any $\varepsilon > 0$, can compute a $1 \pm \varepsilon$ approximation of $z = |Z|$ in time $O(n/\varepsilon^2)$. Cohen's algorithm applies to the more general problem of computing the size of the transitive closure of a graph.

Our main result is that in the special case of sparse matrix product size estimation, we can improve this to expected time $O(n)$ for $\varepsilon > 4/\sqrt[4]{z}$. This means that we have a linear time algorithm for relative error where Cohen's algorithm would use time $O(n\sqrt{z})$.

Approach. To build intuition on the size estimation question, consider the sets $\mathcal{A}_j = \{i \mid (i, j) \in R_1\}$ and $\mathcal{C}_j = \{k \mid (j, k) \in R_2\}$. By definition, $Z = \bigcup_j \mathcal{A}_j \times \mathcal{C}_j$. The size of Z depends crucially on the extent of overlap among the sets $\{\mathcal{A}_j \times \mathcal{C}_j\}_j$. However, the total size of these sets may be much larger than both input and output (see [9]), so any approach that explicitly processes them is unattractive.

The starting point for our improved estimation algorithm is a well-known algorithm for estimating the number of distinct elements in a data streaming context [15]. (We remark that the idea underlying this algorithm is similar to that of Cohen [32].) Our main insight is that this algorithm can be extended such that a set of the form $\mathcal{A}_j \times \mathcal{C}_j$ can be added to the sketch in expected time $O(|\mathcal{A}_j| + |\mathcal{C}_j|)$, i.e., without explicitly generating all pairs. The idea is to use a hash function that is particularly well suited for the purpose: Sufficiently structured to make hash values easy to handle algorithmically, and sufficiently random to make the analysis of sketching accuracy go through.

7.1.1 Motivation

Cohen [33] investigated the use of the size estimation technique in sparse matrix computations. In particular, it can be used to find the optimal order of multiplying sparse matrices, and in memory allocation for sparse matrix computations.

[1] Readers familiar with the database literature may notice that we consider projections that return a set, i.e., that projection is duplicate eliminating. We also observe that any equi-join followed by a projection can be reduced to the case above, having two variables in each relation and projecting away the single join attribute. Thus, there is no loss of generality in considering this minimal case.

In addition, we are motivated by applications in database systems, where size estimation is an important part of query optimisation. Examples of database queries that correspond to boolean matrix products are:

- A query that computes all pairs of people in a social network with a distance 2 connection ("possible friends").
- A query to compute all director-actor pairs who have done at least one movie together.
- In a business database with information on orders, and a categorisation of products into types, compute the relation that contains a tuple (c, p) if customer c has made an order for a product of type p.

As a final example, we consider a fundamental data mining task. Given a list of sets, the famous A-Priori data mining algorithm [6], that we have already cited several times, finds frequent item pairs by counting the number occurrences of item pairs where each single element is frequent. So if $R_1 = R_2$ denotes the relationship between high-support (i.e., frequent) items and sets in which they occur, Z is exactly the pairs of frequent items, and the number of distinct items in Z determines the space usage of A-Priori. Since A-Priori may be very time consuming, it is of interest to establish whether sufficient space is available before choosing the support threshold and running the algorithm.

7.1.2 Further related work

JD sketch

Ganguly et al. [47] previously considered techniques that compute a data structure (a *sketch*) for R_1 and R_2 (individually), such that the two sketches suffice to compute an approximation of z.

Define $n_a = |\{i \mid \exists j.(i,j) \in R_1\}|$ and $n_c = |\{k \mid \exists j.(j,k) \in R_2\}|$. Ganguly et al. show that for any constant c and any β, a sketching method that returns a c-approximation with probability $\Omega(1)$ whenever $z \geq \beta$ must, on a worst-case input, use expected space

$$\Omega(\min(n_1{+}n_2, n_a n_c(n_1/n_a{+}n_2/n_c)/\beta)) = \Omega(\min(n_1{+}n_2, (n_1 n_c{+}n_2 n_a)/\beta)) \text{ bits.}$$

The lower bound proof applies to the case where $n_1 = n_2$, $n_a = n_c$, and $z < n_a + n_c$. We note that [47] claims a stronger lower bound, but their proof does not establish a lower bound above $n_1 + n_2$ bits. Ganguly et al. present a sketch whose worst-case space usage matches the lower bound times polylogarithmic factors (while not stated in [47], the trivial sketch that stores the whole input can be used to nearly match the first term in the minimum).

In Section 7.3 we analyze a simple sketch, previously considered in other contexts by Gibbons [52] and Ganguly and Saha [48]. It similarly matches the above worst-case bound, but the exact space usage is incomparable to that of [47].

The focus of [47] is on space usage, and so the time for updating sketches, and for computing the estimate from two sketches, is not discussed in the paper. Looking at the data structure description we see that the update time grows linearly with the quantity s_1, which is $\Omega(n)$ in the worst case. Also, the sketch

uses a number of summary data structures that are accessed in a random fashion, meaning that the worst case number of I/Os is at least $\Omega(n)$ *unless* the sketch fits internal memory. By the above lower bound we see that keeping the sketch in internal memory is not feasible in general. In contrast, the sketch we consider allows collection and combination of sketches to be done efficiently in linear time and I/O.

Distinct elements and distinct paths estimation

Our work is related in terms of techniques to papers on estimating the number of distinct items in a data stream (see [15] and its references). However, our basic estimation algorithm does not work in a general streaming model, since it crucially needs the ability to access all tuples with a particular value on the join attribute together.

Ganguly and Saha [48] consider the problem of estimating the number of distinct vertex pairs connected by a length-2 path in a graph whose edges are given as a data stream of n edges. This corresponds to size estimation for the special case of *squaring* a matrix (or self-join in database terminology). It is shown that space $\sqrt{n}$ is required, and that space roughly $O(n^{3/4})$ suffices for constant ε (unless there are close to n connected components). The estimation itself is a join-distinct size estimation of a sample of the input having size no smaller than $O(n^{3/4}/\varepsilon^2)$. Using Cohen's estimation algorithm this would require time $O(n^{3/4}/\varepsilon^4)$, so this is $O(n)$ time only for $\varepsilon > 1/\sqrt[16]{n}$.

Join synopses

Acharya et al. [2] proposed so-called *join synopses* that provide a uniform sample of the result of a join. While this can be used to estimate result sizes of a variety of operations, it does not seem to yield efficient estimates of join-project sizes. The reason is that a standard uniform sample is known to be inefficient for estimating the number of distinct values [27]. In addition, Acharya et al. assume the presence of a foreign-key relationship, i.e., that each tuple has at most one matching tuple in the other table(s), which is also known as a *snow flake* schema. Our method has no such restriction.

Distinct sampling

Gibbons [52] considered different samples that can be extracted by a scan over the input, and proposed *distinct samples*, which offer much better guarantees with respect to estimating the number of distinct values in query results. Gibbons shows that this technique applies to single relations, and to foreign key joins where the join result has the same number of tuples as one of the relations. In Section 7.3 we show that the distinct samples, with suitable settings of parameters, can often be used in our setting to get an accurate estimate of $z = |Z|$. The processing of a pair of samples to produce the estimate consists of running the efficient estimation algorithm of Section 7.2 on the samples, meaning that this is time- and I/O-efficient.

7.2 Our algorithm

The task is to estimate the size z of $Z = \pi_{ac}(R_1 \bowtie R_2)$. We may assume that attribute values are $O(\log n)$-bits integers, since any domain can be mapped into this one using hashing, without changing the join result size with high probability. When discussing I/O bounds, B is the number of such integers that fits in a disk block. In linear expected time (by hashing) or sort(n) I/Os we can cluster the relations according to the value of the join attribute b. By initially eliminating input tuples that do not have any matching tuples in the other relation we may assume without loss of generality that $z \geq n/2$.

In what follows, k is a positive integer parameter that determines the space usage and accuracy of our method. The technique used is to compute the kth smallest value v of a hash function $h(x, y)$, for $(x, y) \in Z$. Analogously to the result by Bar-Yossef et al. [15] we can then use $\tilde{z} = k/v$ as an estimator for z. The idea underlying this estimation is that, if v is the value of the k^{th} smallest hash value, the density of values can be assumed to be v/k in the whole interval $[0, 1]$. Hence, if we have $\tilde{z}$ distinct items, it must be $1/\tilde{z} = v/k \Rightarrow \tilde{z} = k/v$.

Our main building block is an efficient iteration over all tuples $(x, \cdot, y) \in R_1 \bowtie R_2$ for which $h(x, y)$ is smaller than a carefully chosen threshold p, and is therefore a candidate for being among the k smallest hash values. The essence of our result lies in how the pairs being output by this iteration are computed in expected linear time. We also introduce a new buffering trick to update the sketch in expected amortised $O(1)$ time per pair. In a nutshell, each time k new elements have been retrieved, they are merged using a linear time selection procedure with the previous k smallest values to produce a new (unordered) list of the k smallest values.

Theorem 7.2.1. *Let $R_1(a, b)$ and $R_2(b, c)$ be relations with n tuples in total, and define $z = |\pi_{ac}(R_1 \bowtie R_2)|$. Let ε, $0 < \varepsilon < \frac{1}{4}$ be given. There are algorithms that run in expected $O(n)$ time on a RAM, and expected $O(sort(n))$ I/Os in the cache-oblivious model, and output a number $\tilde{z}$ such that for $k = 9/\varepsilon^2$:*

- $\Pr[(1-\varepsilon)z < \tilde{z} < (1+\varepsilon)z] \geq 2/3$ *when* $z > k^2$, *and*
- $\Pr[\tilde{z} < (1+\varepsilon)k^2] \geq 2/3$ *when* $z \leq k^2$. ∘

Observe that for $\varepsilon > 4/\sqrt[4]{z}$ we will be in the first case, and get the desired $1 \pm \varepsilon$ approximation with probability $2/3$. The error probability can be reduced from $1/3$ to δ by the standard technique of doing $O(\log(1/\delta))$ runs and taking the median (the analysis follows from a Chernoff bound). We remark that this can be done in such a way that the $O(\log(1/\delta))$ factor affects only the RAM running time and not the number of I/Os. For constant relative error $\varepsilon > 0$ we have the following result:

Theorem 7.2.2. *In the setting of Theorem 7.2.1, if ε is constant there are algorithms that run in expected $O(n)$ time on a RAM, and expected $O(sort(n))$ I/Os in the cache-oblivious model, that output $\tilde{z}$ such that* $\Pr[(1-\varepsilon)z < \tilde{z} < (1+\varepsilon)z] = 1 - O(1/\sqrt{n})$. ∘

The error probability can be reduced to n^{-c} for any desired constant c by running the algorithms $O(c)$ times, and taking the median as above.

7.2.1 Finding pairs

For $\mathcal{B} = \pi_b(R_1) \cup \pi_b(R_2)$ and each $i \in \mathcal{B}$ let $\mathcal{A}_i = \pi_a(\sigma_{b=i}(R_1))$ and $\mathcal{C}_i = \pi_c(\sigma_{b=i}(R_2))$. We would like to efficiently iterate over all pairs $(x, y) \in \mathcal{A}_i \times \mathcal{C}_i$, $i \in \mathcal{B}$, for which $h(x, y)$ is smaller than a threshold p. This is done as follows (see Algorithm 7.1 for pseudocode).

For a set U, let $h_1, h_2 : U \to [0; 1]$ be hash functions chosen independently at random from a pairwise independent family, and define $h : U \times U \to [0; 1]$ by[2]

$$h(x, y) = (h_1(x) - h_2(y)) \bmod 1.$$

It is easy to show that h is also a pairwise independent hash function — a property we will utilize later. Now, conceptually arrange the values of $h(x, y)$ in an $|\mathcal{A}_i| \times |\mathcal{C}_i|$ matrix, and order the rows by increasing values of $h_1(x)$, and the columns by increasing values of $h_2(y)$. Then the values of $h(x, y)$ will decrease (modulo 1) from left to right, and increase (modulo 1) from top to bottom.

For each $i \in \mathcal{B}$, we traverse the corresponding $|\mathcal{A}_i| \times |\mathcal{C}_i|$ matrix by visiting the columns from left to right, and in each column t finding the row $\bar{s}$ with the smallest value of $h(x_{\bar{s}}, y_t)$. Values smaller than p in that column will be found in rows subsequent to $\bar{s}$. When all such values have been output, the search proceeds in column $t + 1$. Notice, that if $h(x_{\bar{s}}, y_t)$ was the minimum value in column t, then the minimum value in column $t+1$ is found by increasing $\bar{s}$ until $h(x_{\bar{s}}, y_{t+1}) < h(x_{(\bar{s}-1) \bmod |\mathcal{A}_i|}, y_{t+1})$. We observe that the algorithm is robust to decreasing the value of the threshold p during execution, in the sense that the algorithm still outputs all pairs with hash value at most p. This algorithm is the same presented in Chapter 3 with some adaptations needed to fit the specific setting. These adaptations are:

- the interval of values we are interested in is $[0, p]$;
- in order to find the flip, we increment the row index instead of decreasing it, because of the hash function h we consider;
- the largest value smaller than p in every column is found starting from the flip, because we want to explicitly produce all the values in a column falling in $[0, p]$.

In order to get the same exact algorithm, it would be sufficient to consider $H_2 = -h_2$ instead of h_2, $g(x, y) = (h_1(x) + H_2(y)) \bmod 1$ instead of h and keep track of the position j in which the largest value smaller than p is found in a column.

7.2.2 Estimating the size

While finding the relevant pairs, we will use a technique that allows us to maintain the k smallest hash values in an unordered buffer instead of using a heap data structure (Lines 14–18 in Algorithm 7.1). In this way we are able to maintain the k smallest hash values in constant amortised time per insertion in the buffer, eliminating the $\log k$ factor implied by the heap data structure.

[2] We observe that this is different from the "composable hash functions" used by Ganguly et al. [47].

```
 1: procedure DisItems(p, ε)
 2:     k := ⌈9/ε²⌉
 3:     F := ∅
 4:     for i ∈ B do
 5:         x := A_i sorted according to h1-value
 6:         y := C_i sorted according to h2-value
 7:         s̄ := 1
 8:         for t := 1 to |C_i| do
 9:             while h(x_s̄, y_t) > h(x_((s̄−1) mod |A_i|), y_t) do
10:                 s̄ := (s̄ + 1) mod |A_i|
11:             end while
12:             s := s̄
13:             while h(x_s, y_t) < p do
14:                 F := F ∪ {(x_s, y_t)}
15:                 if |F| = k then
16:                     (p, S) := Combine(S, F)
17:                     F := ∅
18:                 end if
19:                 s := (s + 1) mod |A_i|
20:             end while
21:         end for
22:     end for
23:     (p, S) := Combine(S, F)
24:     if |S| = k then
25:         return "z̃ = k/p and z̃ ∈ [(1 ± ε)z] with probability 2/3"
26:     else
27:         return "z̃ = k², z ≤ k² with probability 2/3"
28:     end if
29: end procedure

30: procedure Combine(S, F)
31:     v := Rank(h(S) ∪ h(F), k) ▷ Rank(·, k) returns the kth smallest value
32:     S := {x ∈ S ∪ F | h(x) ≤ v}
33:     return (v, S)
34: end procedure
```

Algorithm 7.1: *Pseudocode for the size estimator. The* **While** *loop on Line 9–11 finds $\bar{s}$ such that $h(x_{\bar{s}}, y_t)$ is the minimum in the column. The* **While** *loop on Line 13–20 finds all s where $h(x_s, y_t) < p$. The condition of the* **If** *on Line 15 is verified when the buffer F is filled.*

Let S and F be two unordered sets containing, respectively, the k smallest hash values seen so far (all, of course, smaller than p), and the latest up to k elements seen. We avoid duplicates in S and F (i.e., the sets are kept disjoint) by using a simple hash table to check for membership before insertion. Whenever $|F| = k$ the two sets S and F are combined in order to obtain a new sketch S. This is done by finding the median of $S \cup F$, which takes $O(k)$ time using either deterministic methods (see [42]) or more practical randomized ones [62].

At each iteration the current kth smallest value in S may be smaller than the initial value p, and we use this as a better substitute for the initial value

of p. However, in the analysis below we will upper bound both the running time and the error probability using the initial threshold value p.

7.2.3 Time analysis

We split the time analysis into two parts. One part accounts for iterations of the inner while loop in Lines 13–20, and the other part accounts for everything else. We first consider the RAM model, and then outline the analysis in the cache-oblivious model.

Inner while loop. Observe that for each iteration, one pair (x_s, y_t) is added to F (if it is not already there). For each $t \in \mathcal{C}_i$, $p|\mathcal{A}_i|$ elements are expected to be added since each pair (x_s, y_t) is added with probability p. This means that the expected total number of iterations is $O(p|\mathcal{A}_i||\mathcal{C}_i|)$. Each call to COMBINE costs time $O(k)$, but we notice that there must be at least k iterations between successive calls, since the size of F must go from 0 to k. Inserting a new value into F costs $O(1)$ since the set is not sorted. Hence, the total cost of the inner loop is $O(p|\mathcal{A}_i||\mathcal{C}_i|)$.

Remaining cost. Consider the processing of a single $i \in \mathcal{B}$ in Algorithm 7.1. The initial sorting of hash values can be done in expected time $O(|\mathcal{A}_i| + |\mathcal{C}_i|)$ in the following way. To sort $\mathcal{A}_i$ according to h_1-value we employ a simple bucket sorting method: Create an array of size $|\mathcal{A}_i|$ where entry ℓ points to a "bucket" array that will eventually contain the items $\{x \in \mathcal{A}_i \mid h_1(x) \in [\ell/|\mathcal{A}_i|; (\ell+1)/|\mathcal{A}_i|)\}$. The items are placed one by one in a bucket, taking time linear in the number of items already in the bucket. The total cost for a bucket is quadratic in the number of items it contains at the end. As shown in [45], summing over all buckets this is $O(|\mathcal{A}_i|)$ in expectation if h_1 is pairwise independent. By the same argument, $\mathcal{C}_i$ can be sorted in expected time $O(|\mathcal{C}_i|)$.

For the iteration in Lines 9–11 observe that $h(x_{\bar{s}}, y_t)$ is monotone modulo 1, and we have at most a total of $2|\mathcal{A}_i|$ increments of $\bar{s}$ among all $t \in \mathcal{C}_i$ (since both $h_1(x)$ and $h_2(y)$ map into $[0;1]$ and we consider them in sorted order). Thus, the total number of iterations is $O(|\mathcal{A}_i|)$, and the total cost for each $i \in \mathcal{B}$ is $O(|\mathcal{A}_i| + |\mathcal{C}_i|)$.

The time for the final call to COMBINE is dominated by the preceding cost of constructing S and F.

I/O efficient variant. As for I/O efficiency, notice that a direct implementation of Algorithm 7.1 may cause a linear number of cache misses if $\mathcal{A}_i$ and $\mathcal{C}_i$ do not fit into internal memory. To get an I/O-efficient variant we use a cache-oblivious sorting algorithm [46], sorting R_1 according to $(b, h_1(a))$, and R_2 according to $(b, h_2(c))$, such that the sorting steps for each $i \in \mathcal{B}$ is replaced by one global sorting step.

The rest of the algorithm works directly in a cache-oblivious setting. To see this, notice that it suffices to keep in internal memory the two input blocks that are closest to each of the pointers s, t, and $\bar{s}$. The cache-oblivious model assumes the cache to behave in an optimal fashion, so also in this model there will be $\Omega(B)$ operations between cache misses, and $O(n/B)$ I/Os, expected, in total.

Lemma 7.2.1. *Suppose $R_1(a,b)$ and $R_2(b,c)$ are relations with n tuples in total. Let $p > 0$ and $\varepsilon > 0$ be given. Then Algorithm 7.1 runs in expected $O(n + \sum_i p|\mathcal{A}_i||\mathcal{C}_i|)$ time and $O(1/\varepsilon^2)$ space on a RAM, and can be modified to use expected $O(sort(n))$ I/Os in the cache-oblivious model.* ∘

Choice of threshold p

We would like a value of p that ensures the expected processing time is $O(n)$. At the same time p should be large enough that we expect to reach Line 25 where an exact estimate is returned (except possibly in the case where z is small).

Lemma 7.2.2. *Let $j \in \mathcal{B}$ satisfy $|\mathcal{A}_i||\mathcal{C}_i| \leq |\mathcal{A}_j||\mathcal{C}_j|$ for all $i \in \mathcal{B}$. Then $p = \min(1/k, k/(|\mathcal{A}_j||\mathcal{C}_j|))$ gives an expected $O(n)$ running time for Algorithm 7.1.*

Proof. We argue that for each i, $p|\mathcal{A}_i||\mathcal{C}_i| \leq \max(|\mathcal{A}_i|, |\mathcal{C}_i|)$, which by Lemma 7.2.1 implies running time $O(n + \sum_i p|\mathcal{A}_i||\mathcal{C}_i|) = O(n + \sum_i \max(|\mathcal{A}_i|, |\mathcal{C}_i|)) = O(n)$. Suppose first that $|\mathcal{A}_i||\mathcal{C}_i| \geq k^2$. Then $p = k/(|\mathcal{A}_j||\mathcal{C}_j|)$ and $p|\mathcal{A}_i||\mathcal{C}_i| \leq k \leq \sqrt{|\mathcal{A}_i||\mathcal{C}_i|} \leq \max(|\mathcal{A}_i|, |\mathcal{C}_i|)$. Otherwise, when $|\mathcal{A}_i||\mathcal{C}_i| < k^2$, we have $p \leq 1/k$ and $p|\mathcal{A}_i||\mathcal{C}_i| = |\mathcal{A}_i||\mathcal{C}_i|/k \leq \max(|\mathcal{A}_i|, |\mathcal{C}_i|)$. □

We note that when R_1 and R_2 are sorted according to b, the value of p specified above can be found by a simple scan over both inputs. Our experiments indicate that in practice this initial scan is not needed, see Section 7.4 for details.

7.2.4 Error probability

Theorem 7.2.3. *Let h be a pairwise independent hash function. Suppose we are provided with a stream of elements N with $h(x) < v$ for all $x \in N$. Further, let ε, $0 < \varepsilon < \frac{1}{4}$ be given and assume that $p \geq \min\left(\frac{k}{2z}, \frac{1}{k}\right)$, where $k \geq 9/\varepsilon^2$, and z is the number of distinct items in N. Then Algorithm 7.1 produces an approximation $\tilde{z}$ of z such that*

- $\Pr[(1-\varepsilon)z < \tilde{z} < (1+\varepsilon)z] \geq 2/3$ *for $z > k^2$, and*
- $\Pr[\tilde{z} < (1+\varepsilon)k^2] \geq 2/3$ *for $z \leq k^2$.*

Proof. The error probability proof is similar to the one that can be found in [15], with some differences and extensions. We bound the error probability of three cases: The estimate being smaller or larger than the multiplicative error bound, and the number of obtained samples being too small.

Estimate too large. Let us first consider the case where $\tilde{z} > (1+\varepsilon)z$, i.e. the algorithm overestimates the number of distinct elements. This happens if the stream N contains at least k entries smaller than $k/(1+\varepsilon)z$. For each pair $(a,c) \in Z$ define an indicator random variable $X_{(a,c)}$ as

$$X_{(a,c)} = \begin{cases} 1 & h(a,c) < k/(1+\varepsilon)z \\ 0 & \text{otherwise} \end{cases}$$

That is, we have z such random variables for which the probability of $X_{(a,c)} = 1$ is exactly $k/(1+\varepsilon)z$ and $\mathrm{E}[X_{(a,c)}] = k/(1+\varepsilon)z$. Now define $Y = \sum_{(a,c)\in Z} X_{(a,c)}$

so that $\mathrm{E}[Y] = \mathrm{E}[\sum_{(a,c)\in Z} X_{(a,c)}] = \sum_{(a,c)\in Z} \mathrm{E}[X_{(a,c)}] = k/(1+\varepsilon)$. By the pairwise independence of the $X_{(a,c)}$ we also get $\mathrm{Var}(Y) \leq k/(1+\varepsilon)$. Using Chebyshev's inequality [82] we can bound the probability of having too many pairs reported:

$$\Pr[Y > k] \leq \Pr\left[|Y - \mathrm{E}[Y]| > k - \tfrac{k}{1+\varepsilon}\right] \leq \frac{\mathrm{Var}[Y]}{\left(k - \frac{k}{1+\varepsilon}\right)^2} \leq \frac{k/(1+\varepsilon)}{\left(k - \frac{k}{1+\varepsilon}\right)^2} \leq \tfrac{1}{6}$$

since $k \geq 9/\varepsilon^2$.

Estimate too small. Now, consider the case where $\tilde{z} < (1-\varepsilon)z$ which happens when at most k hash values are smaller than $k/(1-\varepsilon)z$ and at least k hash values are smaller than p. Define $X'_{(a,c)}$ as

$$X'_{(a,c)} = \begin{cases} 1 & h(a,c) < k/(1-\varepsilon)z \\ 0 & \text{otherwise} \end{cases}$$

so that $\mathrm{E}[X'_{(a,c)}] = k/(1-\varepsilon)z < (1+\varepsilon)k/z$. Moreover, with $Y' = \sum_{(a,c)\in Z} X'_{(a,c)}$ we have $\mathrm{E}[Y'] = k/(1-\varepsilon)$, and since the indicator random variables defined above are pairwise independent, we also have $\mathrm{Var}[Y'] \leq \mathrm{E}[Y'] < (1+\varepsilon)k$. Chebyshev's inequality gives:

$$\Pr[Y' < k] \leq \Pr\left[|Y' - \mathrm{E}[Y']| > \tfrac{k}{1-\varepsilon} - k\right] \leq \frac{\mathrm{Var}[Y']}{\left(k - \frac{k}{1+\varepsilon}\right)^2} \leq \frac{(1+\varepsilon)k}{\left(\frac{k}{1-\varepsilon} - k\right)^2} < \tfrac{1}{9}$$

since $k \geq 9/\varepsilon^2$.

Not enough samples. Consider the case where $|S| < k$ after all pairs have been retrieved. In this case the algorithm returns $\beta = k^2$ as an upper bound on the number of distinct elements in the output, and we have two possible situations: either there is actually less than k^2 distinct pairs in the output, in which case the algorithm is correct, or there are more than k^2 distinct elements in the output, in which case it is incorrect. In the latter case, less than k hash values have been smaller than p and the kth smallest value v is therefore larger than p. Define $X''_{(a,c)}$ as

$$X''_{(a,c)} = \begin{cases} 1 & h(a,c) < p \\ 0 & \text{otherwise} \end{cases}$$

and let again $Y'' = \sum_{(a,c)\in Z} X''_{(a,c)}$. It results that $\mathrm{E}[X''_{(a,c)}] = p$ and $\mathrm{E}[Y''] = zp$, and because of pairwise independence of $X''_{(a,c)}$, also $\mathrm{Var}[Y''] \leq \mathrm{E}[Y'']$. Using Chebyshev's inequality and remembering that $z > k^2$ in this case we have:

$$\Pr[Y'' < k] \leq \Pr[|Y'' - \mathrm{E}[Y'']| > zp - k] \leq \frac{zp}{(zp-k)^2} \leq \frac{zp}{\left(\frac{1}{2}zp\right)^2} \leq 8/k \leq 1/18.$$

using that $k \geq 9/\varepsilon^2 \geq 144$.

In conclusion, the probability that the algorithm fails to output an estimate within the given limits is at most $1/6 + 1/9 + 1/18 = 1/3$. □

For the proof of Theorem 7.2.2 we observe that in the above proof, if ε is constant the error probability is $O(1/k)$. Using $k = \sqrt{n}$ we get linear running time and error probability $O(1/\sqrt{n})$.

Realization of hash functions

We have used the idealised assumption that hash values were real numbers in $(0;1)$. Let $m = n^3$. To get an actual implementation we approximate (by rounding down) the real numbers used by rational numbers of the form i/m, for integer i. This changes each hash value by at most $2/m$. Now, because of the way hash values are computed, the probability that we get a different result when comparing two real-valued hash values and two rational ones is bounded by $2/m$. Similarly, the probability that we get a different result when looking up a hash value in the dictionary is bounded by $2k/m$. Thus, the probability that the algorithm makes a different decision based on the approximation, in any of its steps, is $O(kn/m) = o(1)$. Also, for the final output the error introduced by rounding is negligible.

7.3 Distinct sketches

A well-known approach to size estimation in, described in generality by Gibbons [52] and explicitly for join-project operations in [48, 9], is to sample random subsets $R_1' \subseteq R_1$ and $R_2' \subseteq R_2$, compute $Z' = \pi_{ac}(R_1' \bowtie R_2')$, and use the size of Z' to derive an estimate for z. This is possible if $R_1' = \sigma_{a \in S_a}(R_1)$, where $S_a \subseteq \pi_a(R_1)$ is a random subset where each element is picked independently with probability p_1, and similarly $R_2' = \sigma_{c \in S_c}(R_2)$, where $S_c \subseteq \pi_c(R_2)$ includes each element independently with probability p_2. Then $z' = |Z'|/(p_1 p_2)$ is an unbiased estimator for z. The samples can be obtained in small space using hash functions whose values determine which elements are picked for S_a and S_c. The value $|Z'|$ can be approximated in linear time using the method described in Section 7.2 if the samples are sorted — otherwise one has to add the cost of sorting. In either case, the estimation algorithm is I/O-efficient.

Below we analyse the variance of the estimator z', to identify the minimum sampling probability that introduces only a small relative error with good probability. The usual technique of repetition can be used to reduce the error probability. Recall that we have two relations with n_1 and n_2 tuples, respectively, and that n_a and n_c denotes the number of distinct values of attributes a and c, respectively. Our method will pick samples R_1' and R_2' of expected size s from each relation, where $s = p_1 n_1 = p_2 n_2$ is a parameter to be specified.

Theorem 7.3.1. *Let R_1' and R_2' be samples of size s, obtained as described above. Then $z' = |\pi_{ac}(R_1' \bowtie R_2')|/(p_1 p_2)$ is a $1 \pm \varepsilon$ approximation of $z = |\pi_{ac}(R_1 \bowtie R_2)|$ with probability $5/6$ if $z > \beta$, where $\beta = \frac{14}{\varepsilon^2}\left(\frac{n_c n_1 + n_a n_2}{s}\right)$. If $z \leq \beta$ then $z' < (1+\varepsilon)\beta$ with probability $5/6$.* ∘

7.3.1 Analysis of variance

To arrive at a sufficient condition that z' is a $1 \pm \varepsilon$ approximation of z with good probability, we analyse its variance. To this end define $Z_{i\cdot} = \{j \mid (i,j) \in Z\}$, $Z_{\cdot j} = \{i \mid (i,j) \in Z\}$, and let

$$X_i = \begin{cases} 1 - p_1, & \text{if } i \in S_a \\ -p_1, & \text{otherwise} \end{cases} \qquad Y_j = \begin{cases} 1 - p_2, & \text{if } j \in S_c \\ -p_2, & \text{otherwise} \end{cases} .$$

By definition of S_a, $\mathrm{E}[X_i] = \Pr[i \in S_a](1-p_1) - \Pr[i \notin S_a]p_1 = 0$. Similarly, $\mathrm{E}[Y_i] = 0$. We have that $(i,j) \in Z'$ if and only if $(i,j) \in Z$ and $(i,j) \in S_a \times S_c$. This means that $z'p_1p_2 = \sum_{(i,j)\in Z}(X_i+p_1)(Y_j+p_2)$. By linearity of expectation, $\mathrm{E}[(X_i+p_1)(Y_j+p_2)] = p_1p_2$, and we can write the variance of $z'p_1p_2$, $\mathrm{Var}(z'p_1p_2)$ as

$$\mathrm{E}\left[\left(\sum_{(i,j)\in Z}((X_i+p_1)(Y_j+p_2)-p_1p_2)\right)^2\right].$$

Expanding the product and using linearity of expectation, we get

$$\begin{aligned}\mathrm{Var}(z'p_1p_2) &= \sum_{(i,j)\in Z}\sum_{(i,j')\in Z}\mathrm{E}\left[X_i^2p_2^2\right] + \sum_{(i,j)\in Z}\sum_{(i',j)\in Z}\mathrm{E}\left[Y_j^2p_1^2\right] + \sum_{(i,j)\in Z}\mathrm{E}\left[X_i^2Y_j^2\right] \\ &= \sum_{i\in\mathcal{A}}\sum_{j,j'\in Z_{i\cdot}}p_2^2\,\mathrm{E}\left[X_i^2\right] + \sum_{j\in\mathcal{C}}\sum_{i,i'\in Z_{\cdot j}}p_1^2\,\mathrm{E}\left[Y_i^2\right] + z\,\mathrm{E}\left[X_i^2\right]\mathrm{E}\left[Y_i^2\right]\end{aligned}$$

Since $\mathrm{E}\left[X_i^2\right] = p_1(1-p_1)^2 + (1-p_1)(-p_1)^2 = p_1 - p_1^2 < p_1$, and similarly $\mathrm{E}\left[Y_j^2\right] < p_2$ we can upper bound $\mathrm{Var}(z')$ as follows:

$$\begin{aligned}\mathrm{Var}(z') &= (p_1p_2)^{-2}\,\mathrm{Var}(z'p_1p_2) \\ &< (p_1p_2)^{-2}\Big(\sum_{i\in\mathcal{A}}\sum_{j,j'\in Z_{i\cdot}}p_1p_2^2 + \sum_{j\in\mathcal{C}}\sum_{i,i'\in Z_{\cdot j}}p_1^2p_2 + z\,p_1p_2\Big) \\ &\le (p_1p_2)^{-2}\left(n_cz\,p_1p_2^2 + n_az\,p_1^2p_2 + z\,p_1p_2\right) \\ &= \left(n_c/p_1 + n_a/p_2 + (p_1p_2)^{-1}\right)\,z\ .\end{aligned}$$

7.3.2 Sufficient sample size

We are ready to derive a bound on the probability that z' deviates significantly from z. Choose $0 < \varepsilon < 1$. Since $z = \mathrm{E}[z']$ Chebyshev's inequality says

$$\Pr[|z'-z| > \varepsilon z] < \frac{\mathrm{Var}(z')}{(\varepsilon z)^2} \le \left(n_c/p_1 + n_a/p_2 + (p_1p_2)^{-1}\right)/(\varepsilon^2 z).$$

This can equivalently be expressed in terms of the sample size s, since $p_1 = s/n_1$ and $p_2 = s/n_2$:

$$\Pr[|z'-z| > \varepsilon z] < \left(n_cn_1 + n_an_2 + n_1n_2/s\right)/(s\varepsilon^2 z).$$

We seek a sufficient condition on s that the above probability is bounded by some constant $\delta < \frac{1}{2}$ (e.g. $\delta = 1/6$). In particular it must be the case that $n_1n_2/(s^2\varepsilon^2 z) < \delta$, which implies $s > \sqrt{n_1, n_2/(\delta z)} \ge \sqrt{n_1, n_2/(\delta n_a n_c)}$. Hence, using the arithmetic-geometric inequality:

$$n_1n_2/s < \sqrt{n_cn_1n_an_2\delta} \le (n_cn_1 + n_an_2)/(2\sqrt{\delta}).$$

In other words, it suffices that

$$\frac{(n_cn_1 + n_an_2)\,(1 + (2\sqrt{\delta})^{-1})}{s\varepsilon^2 z} < \delta \iff s > \left(\frac{n_cn_1 + n_an_2}{z}\right)\left(\frac{1 + (2\sqrt{\delta})^{-1}}{\varepsilon^2\delta}\right).$$

One apparent problem is the chicken-egg situation: z is not known in advance. If a lower bound on z is known, this can be used to compute a sufficient sample size. Alternatively, if we allow a larger relative error whenever $z \leq \beta$ we may compute a sufficient value of s based on the assumption $z \geq \beta$. Whenever $z < \beta$ we then get the guarantee that $z' < (1+\varepsilon)\beta$ with probability $1-\delta$. Theorem 7.3.1 follows by fixing s and solving for β.

Optimality

For constant ε and δ our upper bound matches the lower bound of Ganguly et al. [47] whenever this does not exceed $n_1 + n_2$. It is trivial to achieve a sketch of size $O((n_1+n_2)\log(n_1+n_2))$ bits (simply store hash signatures for the entire relations). We also note that the lower bound proof in [47] uses certain restrictions of parameters ($n_1 = n_2$, $n_a = n_c$, and $z < n_a + n_c$), so it may be possible to do better in some settings.

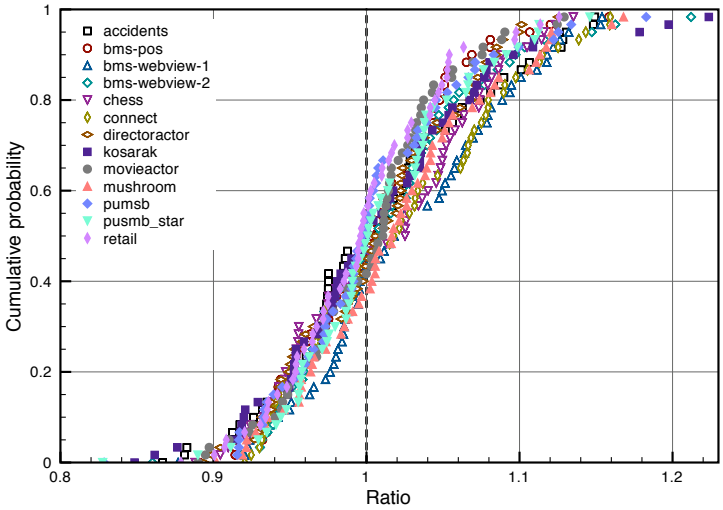
7.4 Experiments

We have run our algorithm on most of the datasets from the Frequent Itemset Mining Implementations (FIMI) Repository[3] together with some datasets extracted from the Internet Movie Database (IMDB). Each dataset represents a single relation, and motivated by the A-Priori space estimation example in the introduction, we perform the size estimation on self-joins of these relations. Table 7.1 displays the size of each dataset together with the number of distinct a- and c-values.

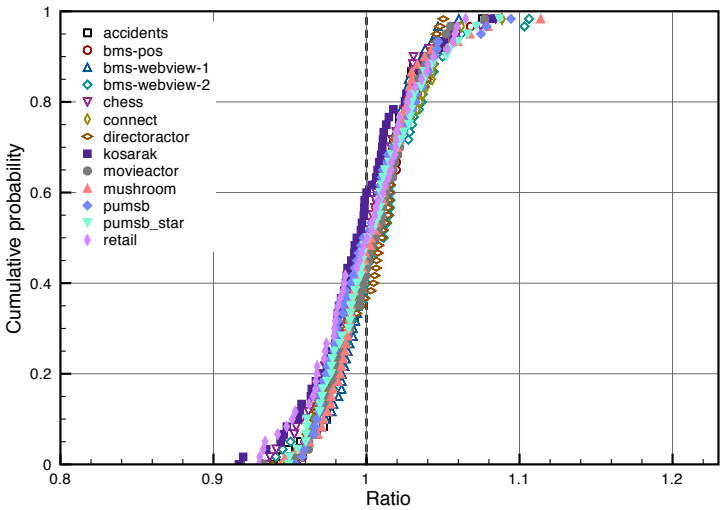
Instance	z	$n_a\,(=n_c)$	$\varepsilon_{0.1}$	$\varepsilon_{0.01}$
Accidents	$94 \cdot 10^3$	468	1.18	3.73
BMS-POS	$760 \cdot 10^3$	1,657	0.78	2.47
BMS-WebView-1	$128 \cdot 10^3$	497	1.04	3.29
BMS-webView-2	$1.45 \cdot 10^6$	3,340	0.80	2.54
Chess	$5.24 \cdot 10^3$	75	2.00	6.33
Connect	$13.8 \cdot 10^3$	129	1.62	5.12
DirectorActor	$734 \cdot 10^6$	50,645	0.14	0.44
Kosarak	$66.2 \cdot 10^6$	41,270	0.42	1.32
MovieActor	$111 \cdot 10^6$	51,226	0.36	1.14
Mushroom	$7.17 \cdot 10^3$	119	2.16	6.82
Pumsb	$1.07 \cdot 10^6$	2,113	0.74	2.35
Pumsb_star	$967 \cdot 10^3$	2,088	0.78	2.46
Retail	$7.19 \cdot 10^6$	16,470	0.80	2.53

Table 7.1: *Characteristics of the used datasets. The rightmost middle column displays the size $n_a = |\pi_a(R_1)|$ (which in this case is equals $n_c = |\cup\pi_c(R_2)|$). The two rightmost columns display the theoretical error as described in Theorem 7.3.1, for $p_1 = p_2 = 0.1$ and $p_1 = p_2 = 0.01$, respectively. These theoretical error bounds, which hold with probability $5/6$, are significantly larger than the actual observed errors in Figure 7.2.*

[3] http://fimi.cs.helsinki.fi

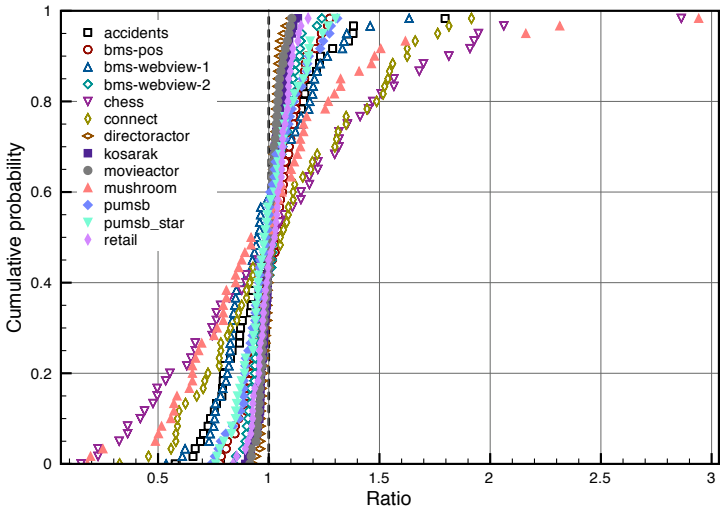


(a) $k = 256$

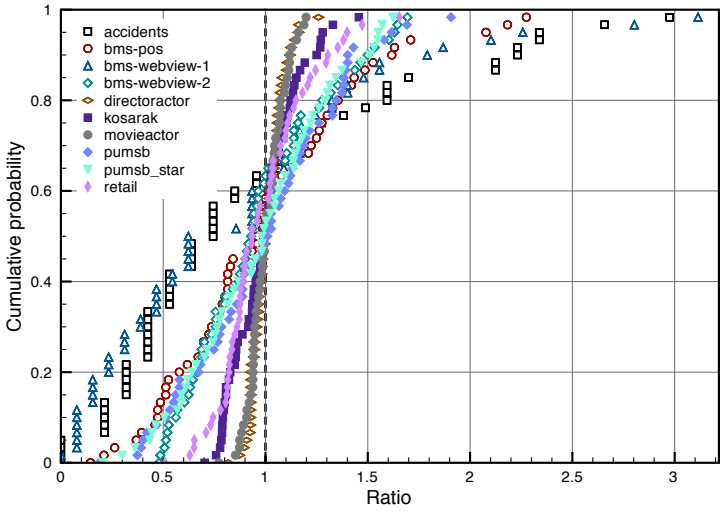


(b) $k = 1024$

Figure 7.1: *The cumulative distribution functions for* $k = 256$ *and* $k = 1024$*. It is seen that* $k = 1024$ *yields a more precise estimate than* $k = 256$ *with 2/3 of the estimates being within* 4% *and* 10% *of the exact size, respectively.*



(a) $k = 1024$, $p_1 = p_2 = 0.1$



(b) $k = 1024$, $p_1 = p_2 = 0.01$

Figure 7.2: *Plots for sampling with probability 10% and 1%. If the sampling probability is too small, no elements at all may reach the sketch and in these cases we are not able to return an estimate. Instances with no estimates have been left out of the graph.*

Rather than selecting h_1 and h_2 from an arbitrary pairwise independent family, we store functions that map the attribute values to fully random and independent values of the form $d/2^{64}$, where d is a 64 bit random integer formed by reading 64 random bits from the Marsaglia Random Number CDROM[4].

We have chosen an initial value of $p = 1$ for our tests in order to be certain to always arrive at an estimate. In most cases we observed that p quickly decreases to a value below $1/k$ anyway. But as the sampling probability decreases, the probability that the sketch will never be filled increases, implying that we will not get a linear time complexity with an initial value of $p = 1$. In the cases where the sketch is not filled, we report $|F|/(p_1 p_2)$ as the estimate, where $|F|$ is the number of elements in the buffer.

Tests have been performed for $k = 256$ and $k = 1024$. In each test, 60 independent estimates were made and compared to the exact size of the join-project. By sorting the ratios "estimate"/"exact size" we can draw the cumulative distribution function for each instance that, for each ratio-value on the x-axis, displays on the y-axis the probability that an estimate will have this ratio or less. Figure 7.1 shows plots for $k = 256$ and $k = 1024$. In Table 7.2 we compare the theoretical error ε with observed error for 2/3 of the results. As seen, the observed error is smaller than the theoretical upper bound.

In Figure 7.2 we perform sampling with 10% and 1% probability, as described in Section 7.3. Again, the samples are chosen using truly random bits. The variance of estimates increases as the probability decreases, but increases more for smaller than for larger instances. If the sampling probability is too small, no elements at all may reach the sketch and in these cases we are not able to return an estimate. As seen, the observed errors in the figure are significantly smaller than the theoretical errors seen in Table 7.1.

k	ϵ	Observed ε
256	0.188	0.1
1024	0.094	0.04

Table 7.2: *The theoretical error bound is $\varepsilon = \sqrt{9/k}$ as stated Theorem 7.2.3. The observed error in Figure 7.1, however, is significantly less.*

7.5 Conclusion

We have presented improved algorithms for estimating the size of boolean matrix products, for the first time allowing $o(1)$ relative error to be achieved in linear time. An interesting open problem is if this can be extended to transitive closure in general graphs, and/or to products of more than two matrices.

[4] http://www.stat.fsu.edu/pub/diehard/

Chapter 8

Epilogue

Sampling from implicit sets is not in general an entirely new technique. As a matter of fact the idea has been used for rather significant research, and some of these findings have been regarded as influential enough to be awarded prizes.

It is important to highlight the work of Dyer, Frieze and Kannan [44], where a technique similar to ours, in the fact that the sampling happens on an implicit object, is used in order to provide an approximation algorithm for computing the volume of convex bodies. Computing the volume of convex bodies takes time that is exponential in n, where n is the number of dimensions. The cited paper proposed a randomized approximation algorithm that computes an $\epsilon-$approximation of the volume with high probability, in time that is polynomial in the number of dimensions n. Notice that it would be infeasible to achieve an approximation even within a polynomial factor in deterministic polynomial time, since a hardness result is contained in [16]. In order to accomplish the result, the algorithm performs a random walk over an implicitly defined undirected graph. The graph models cubes that try to cover the space of the body, and an estimate of the body volume is derived from the size of the cubes covered by the random walk. The random walk will hence visit a number of vertices of the graphs that are the cubes, and provide an estimate after a sufficient number of steps have been taken. In this way, it is not necessary to draw the entire graph, that is, producing all the cubes, but only the portion of graph, hence the portion of cubes, involved in the walk. A polynomial number of such entities is necessary to the algorithm to achieve the desired result.

In most cases, the difference from our approach stands in a fundamental point: the complexity class in which the problems we address lie in. For the convex body, all known deterministic algorithms are exponential in the number of dimensions of the space, and the approximation randomized algorithm reduces the running time to polynomial. Almost all the problems that we address are solvable in polynomial time, usually quadratic time, and the algorithms we present reduce the running time to quasi-linear or linear. This means that the problem of and the structural approach to the sampling faces entirely different constraints and opportunities.

Moreover, the sampling techniques we use are highly innovative for the problems we have presented. Sometimes (see Chapter 6), sampling is an entirely new approach to the framework we applied it. Our techniques offer the appealing characteristic of avoiding the generation of candidate solutions to be selected,

in a second phase, to be output. The candidate generation phase, in particular when carried out in the early stages of many algorithms, operates on the whole input, and for this reason produces a lot of intermediate artefacts; this production is expensive from both a time and a space perspective.

Possible evolutions

BiSam family of algorithms.

A rather natural extension of the techniques presented would be addressing itemsets of size larger than 2. At first glance, the sampling methods we presented cannot be immediately extended in this sense, and using it as a first phase of a multiphase algorithm would make the algorithm fall in the class of candidate generating algorithms, which is not desirable, as explained before.

One possible approach is to use geometric properties of vectors, hence matrix multiplication techniques, in order to get estimates of some of the measure functions like cosine. Additionally similarity functions for itemsets of size larger than two are not so frequent.

Another interesting direction to follow is finding more hardness results, taking into account various possible characteristics of the input. A very natural development would be, for instance, extending the lower bound of Theorem 4.1.1 on page 49, in order to take into account the random order of transactions.

Graph mining

Graph mining offers a wide space of manoeuvre in order for our sampling techniques to be used.

Many counting problems seem to be good candidates for reducing polynomial time complexities to linear or quasi-linear. Counting $k-$cliques in a graph can be one such problem. The obvious worst case time for an exact algorithm would be $n^{O(k)}$, that is particularly expensive when k is large.

A $k-$clique is a vertex connected to all nodes in a $(k-1)-$clique, so an approach to estimating the number of cliques could exploit this recursive structure in order to sample a number of cliques that is not too large, using a suitable sampling probability. If one is able to sample with a probability that is independent enough and uniform, or almost uniform, a conclusion can be derived on the total expected number of $k-$cliques in the graph. In essence the algorithm should try to take samples from the set of subgraphs of size k without explicitly producing that set.

Another problem similar in spirit can be estimating the number of cliques of low degree nodes in a dense graph. This kind of graphs are interesting for the many real world structures that they are able to represent; a meaningful example with this respect are social networks.

Matrix multiplication

As pointed out, the problem is widely general, so extending and deepening this technique is a rather appealing path. It remains, as a matter of fact, to be understood whether the algorithm: (i) can be adapted and generalized in order to address the problem of transitive closure for general graphs; (ii) can be extended and adapted to the case of multiplication of multiple matrices.

Bibliography

[1] Eivind Abusland and Matyas Markovics. Implementing and evaluating a sampling-based approach to association mining on mapreduce. Master's thesis, IT University of Copenhagen, 2011.

[2] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, volume 28(2) of *SIGMOD Record*, pages 275–286. ACM, 1999.

[3] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.

[4] Charu C. Aggarwal and Philip S. Yu. A new framework for itemset generation. In *Proceedings of the ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS '98)*, pages 18–24. ACM Press, 1998.

[5] Rakesh Agrawal, Manish Mehta, John C. Shafer, Ramakrishnan Srikant, Andreas Arning, and Toni Bollinger. The quest data mining system. In *Proceedings of the 2nd International Conference of Knowledge Discovery and Data Mining (KDD '96)*, pages 244–249. AAAI Press, 1996.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. pages 487–499. Morgan Kaufmann, 1994.

[7] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[8] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques; Proceedings of the 13th International Workshop, (APPROX '10), and 14th International Workshop, (RANDOM '10)*, volume 6302 of *Lecture Notes in Computer Science*, pages 406–419. Springer, 2010.

[9] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of Database Theory - 12th International Conference (ICDT '09)*, volume 361 of *ACM International Conference Proceeding Series*, pages 121–126. ACM, 2009.

[10] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, January 2007.

[11] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*, pages 918–929. ACM, 2006.

[12] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[13] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti Spaccamela, and Marco Protasi. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties*. Springer Verlag, 1999.

[14] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '02)*, pages 1–16. ACM, 2002.

[15] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques (RANDOM '02)*, pages 1–10. Springer-Verlag, 2002.

[16] Imre Bárány and Zoltán Füredi. Computing the volume is difficulte. *Discrete & Computational Geometry*, 2:319–326, 1987.

[17] Christian Borgelt. Recursion pruning for the apriori algorithm. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '04)*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[18] Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '99)*, pages 254–260. ACM Press, 1999.

[19] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):265–276, 1997.

[20] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data (SIGMOD '97)*, volume 26(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 255–264. ACM Press, 1997.

[21] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[22] Andrea Campagna and Rasmus Pagh. Finding associations and computing similarity via biased pair sampling. *Knowledge and Information Systems*, pages 1–22. 10.1007/s10115-011-0428-y.

[23] Andrea Campagna and Rasmus Pagh. Finding associations and computing similarity via biased pair sampling. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09)*, pages 61–70. IEEE Computer Society, 2009.

[24] Andrea Campagna and Rasmus Pagh. On finding frequent patterns in event sequences. In *Proceedings of the 10th IEEE International Conference on Data Mining (ICDM '10)*, pages 755–760. IEEE Computer Society, 2010.

[25] Andrea Campagna and Rasmus Pagh. On finding similar items in a stream of transactions. In *Proceedings of the 10th IEEE International Conference on Data Mining Workshops (ICDMW '10)*, pages 121–128. IEEE Computer Society, 2010.

[26] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. Robust lower bounds for communication and stream computation. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC '08)*, pages 641–650. ACM, 2008.

[27] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS '00)*, pages 268–279. ACM, 2000.

[28] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[29] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th annual ACM symposium on Theory of computing (STOC '02)*, pages 380–388. ACM, 2002.

[30] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, page 5. IEEE Computer Society, 2006.

[31] Yen-Liang Chen and Ya-Han Hu. Constraint-based sequential pattern mining: The consideration of recency and compactness. *Decision Support Systems*, 42(2):1203 – 1215, 2006.

[32] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[33] Edith Cohen. Structure prediction and computation of sparse matrix products. *J. Comb. Optim*, 2(4):307–332, 1998.

[34] Edith Cohen, Mayur Datar, Shinji Fujiwara, Aristides Gionis, Piotr Indyk, Rajeev Motwani, Jeffrey D. Ullman, and Cheng Yang. Finding interesting associations without support pruning. *IEEE Trans. Knowl. Data Eng*, 13(1):64–78, 2001.

[35] Edith Cohen and David D. Lewis. Approximating matrix multiplication for pattern recognition tasks. *J. Algorithms*, 30(2):211–252, 1999.

[36] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[37] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *PVLDB*, 1(2):1530–1541, 2008.

[38] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Exponentially decayed aggregates on data streams. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM '09)*, pages 1379–1381. IEEE, 2008.

[39] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *Proceedings of the SIAM International Conference on Data Mining*, 2005.

[40] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Trans. Database Syst.*, 30(1):249–278, 2005.

[41] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA '02)*, volume 2461 of *Lecture Notes in Computer Science*, pages 348–360. Springer, 2002.

[42] Dorit Dor and Uri Zwick. Selecting the median. In *Proceedings of the 6th annual ACM-SIAM Symposium on Discrete algorithms (SODA '95)*, pages 28–37. SIAM, 1995.

[43] Devdatt Dubhashi and Desh Ranjan. Balls and bins: a study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.

[44] Martin E. Dyer, Alan M. Frieze, and Ravi Kannan. A random polynomial time algorithm for approximating the volume of convex bodies. *J. ACM*, 38(1):1–17, 1991.

[45] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

[46] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science, (FOCS '99)*, pages 285–298. IEEE Computer Society, 1999.

[47] Sumit Ganguly, Minos Garofalakis, Amit Kumar, and Rajeev Rastogi. Join-distinct aggregate estimation over update streams. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems (PODS '05)*, pages 259–270. ACM, 2005.

[48] Sumit Ganguly and Barna Saha. On estimating path aggregates over streaming graphs. In *Proceedings of 17th International Symposium on Algorithms and Computation, (ISAAC '06)*, volume 4288 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2006.

[49] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[50] Karolien Geurts, Geert Wets, Tom Brijs, and Koen Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board*, 2003.

[51] Fosca Giannotti, Mirco Nanni, Dino Pedreschi, and Fabio Pinelli. Mining sequences with temporal annotations. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*, pages 593–597. ACM, 2006.

[52] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 541–550. Morgan Kaufmann Publishers, 2001.

[53] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Report of fimi'03. *ACM SIGKDD Explorations*, 6(1):109–117, 2004.

[54] Bart Goethals and Mohammed Javeed Zaki, editors. *Proceedings of the IEEE ICDM 2003 Workshop on Frequent Itemset Mining Implementations (FIMI '03)*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[55] Oded Goldreich. On post-modern cryptography. Cryptology ePrint Archive, Report 2006/461, 2006. `http://eprint.iacr.org/`.

[56] Oded Goldreich. *Computational complexity - a conceptual perspective*. Cambridge University Press, 2008.

[57] Oded Goldreich. In a world of p=bpp. In *Studies in Complexity and Cryptography*, volume 6650 of *Lecture Notes in Computer Science*, pages 191–232. Springer, 2011.

[58] Sudipto Guha and Andrew McGregor. Stream order and order statistics: Quantile estimation in random-order streams. *SIAM Journal on Computing*, 38(5):2044–2059, 2009.

[59] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques, 2nd edition*. Morgan Kaufmann, 2006.

[60] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov*, 8(1):53–87, 2004.

[61] Sherri K. Harms, Jitender S. Deogun, and Tsegaye Tadesse. Discovering sequential association rules with constraints and time lags in multiple sequences. In *Proceedings of the 13th International Symposium on Foundations of Intelligent Systems (ISMIS '02)*, pages 432–441. Springer-Verlag, 2002.

[62] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4(7):321–322, 1961.

[63] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[64] Piotr Indyk. A small approximately min-wise independent family of hash functions. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 454–456, 1999.

[65] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 618–625, 1997.

[66] Nan Jiang and Le Gruenwald. Research issues in data stream association rule mining. *SIGMOD Record*, 35(1):14–19, 2006.

[67] Mahesh V. Joshi, , George Karypis, and Vipin Kumar. A universal formulation of sequential patterns. In *Proceedings of the KDD'2001 workshop on Temporal Data Mining*, 2001.

[68] M. Karnaugh. The Map Method for Synthesis of Combinational Logic Circuits. *IEEE Trans. American Inst. Electrical Eng. pt. I*, 72(9):593–599, 1953.

[69] R. M. Karp and M. O. Rabin. Efficient randomized pattern–matching algorithms. *IBM Journal of Research and Development*, 32:249–260, 1987.

[70] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[71] Neal Koblitz. The uneasy relationship between mathematics and cryptography. *Notices of the American Mathematical Society*, 54, 2007.

[72] Ron Kohavi, Carla Brodley, Brian Frasca, Llew Mason, and Zijian Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[73] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, New York, 1997.

[74] Young-Koo Lee, Won-Young Kim, Y. Dora Cai, and Jiawei Han. Comine: Efficient mining of correlated patterns. In *Proceedings of the IEEE International Conference on Data Mining (ICDM '03)*, pages 581–584. IEEE Computer Society, 2003.

[75] Andrzej Lingas. A fast output-sensitive algorithm for boolean matrix multiplication. In *Proceedings of the 17th European Symposium on Algorithms (ESA '09)*, volume 5757 of *Lecture Notes in Computer Science*, pages 408–419. Springer, 2009.

[76] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 346–357. Morgan Kaufmann Publishers, 2002.

[77] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov*, 1(3):259–289, 1997.

[78] Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki, editors. *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI '04)*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[79] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of Database Theory - 10th International Conference (ICDT '05)*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[80] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. Technical Report 23, University of California, Santa Barbara, USA, 2005.

[81] Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[82] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[83] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[84] Edward Omiecinski. Alternative interest measures for mining associations in databases. *IEEE Trans. Knowl. Data Eng*, 15(1):57–69, 2003.

[85] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[86] Jong Soo Park, Ming-Syan Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):175–186, 1995.

[87] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*, page 215. IEEE Computer Society, 2001.

[88] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB '95)*, pages 432–444. Morgan Kaufmann Publishers, 1995.

[89] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Advances in Database Technology - Proceedings of the 5th International Conference on Extending Database Technology (EDBT '96)*, volume 1057 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 1996.

[90] Hannu Toivonen. Sampling large databases for association rules. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB '96)*, pages 134–145. Morgan Kaufmann Publishers, 1996.

[91] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[92] Xindong Wu, Chengqi Zhang, and Shichao Zhang. Efficient mining of both positive and negative association rules. *ACM Trans. Inf. Syst.*, 22:381–405, 2004.

[93] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. In *Proceedings of the 25th International Conference on Data Engineering, (ICDE '09)*, pages 916–927. IEEE, 2009.

[94] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web, (WWW '08)*, pages 131–140. ACM, 2008.

[95] Jeffrey Xu Yu, Zhihong Chong, Hongjun Lu, Zhenjie Zhang, and Aoying Zhou. A false negative approach to mining frequent itemsets from high speed transactional data streams. *Inf. Sci.*, 176(14):1986–2015, 2006.

[96] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.

[97] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[98] Shichao Zhang, Xindong Wu, Chengqi Zhang, and Jingli Lu. Computing the minimum-support for mining frequent patterns. *Knowl. Inf. Syst.*, 15(2):233–257, 2008.

[99] Q. Zhao and S.S. Bhowmick. Sequential pattern mining: a survey. Technical report, School of Computer Engineering, Nanyang Technological University, Singapore, 2003.

[100] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB '02)*, pages 358–369. Morgan Kaufmann, 2002.