Building Space-Efficient Inverted Indexes on Low-Cardinality Dimensions[⋆]

Vasilis Spyropoulos and Yannis Kotidis

Athens University of Economics and Business,
76 Patission Street, Athens, Greece.
{vasspyrop,kotidis}@aueb.gr

Abstract. Many modern applications naturally lead to the implementation of inverted indexes for effectively managing large collections of data items. Creating an inverted index on a low cardinality data domain results in replication of data descriptors, leading to increased storage overhead. For example, the use of RFID or similar sensing devices in supply-chains results in massive tracking datasets that need effective spatial or spatio-temporal indexes on them. As the volume of data grows proportionally larger than the number of spatial locations or time epochs, it is unavoidable that many of the resulting lists share large subsets of common items. In this paper we present techniques that exploit this characteristic of modern big-data applications in order to losslessly compress the resulting inverted indexes by discovering large common item sets and adapting the index so as to store just one copy of them. We apply our method in the supply chain domain using modern big-data tools and show that our techniques in many cases achieve compression ratios that exceed 50%.

1 Introduction

Many applications implement auxiliary indexes of the form $id \leftarrow list(itemId)$, often referred to as inverted indexes. For example, such indexes are used in performing keyword search in a collection of documents ($word \leftarrow list(documentId)$). In supply-chain applications equipped with RFID tracking technology tagged objects are recorded while passing through a check point (reader's location) in the supply-chain [1, 2]. Inverted indexes can be used to implement spatial ($locationId \leftarrow list(tagId)$), temporal ($epochId \leftarrow list(tagId)$) or spatio-temporal ($(locationId, epochId) \leftarrow list(tagId)$) indexes. When inverted indexes are build on a low cardinality data domain (e.g. *locationId*, *epochId*), the resulting lists tend to share a possibly large number of common item references (e.g. *tagIds*). A straightforward implementation of these lists will result in excessive replication of item references, increasing the space overhead of the inverted index. While

[⋆] This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: RECOST

there is a wealth of techniques that compress inverted indexes at binary level, there is an opportunity to achieve significant compression ratios by looking at the actual content of the lists. The existing methods usually represent the index lists as a sorted sequence of integers and transform it into more compressible form by using the distances between the integers (d-gaps) and applying integer coding techniques such as *Variable Byte*, *Golomb* or *Rice* codes. Our technique takes place at the information level and, thus, can be safely applied before any low level compression of the index. In brief, our method is aiming at finding intersections of the lists and materialize them as new lists, which we call *derived lists*. When a large portion of the original lists are replaced by references to derived lists, we are able to reduce the space of the index since a single copy of item ids is referred by many lists. We identify the following challenges:

Performing Intersections of Large Lists: While checking combinations of lists in order to generate the candidate derived lists we need to have a way so as to efficiently compute the size of their intersection. Since the actual computation of a large number of intersections of lists, each containing millions of records, is a costly operation we utilize approximation techniques. In a first phase we reduce the number of intersections to compute by testing only combinations of similar lists stored in the same bucket among buckets populated by the use of the minhashing/LSH technique. Then, instead of computing the actual intersections, we estimate their size reusing the already computed minhash signatures of the lists and a novel adaptation of the inclusion-exclusion principle for the Jaccard similarity measure, as we discuss in Section 2.

Frequent Itemset Mining for Candidate Generation: A derived list is actually a set of frequent items in the index. Unfortunately, a straightforward implementation of the well known Frequent Itemset Mining *A-priori* algorithm [3] will not work in our setting. In applications like the ones we described the goal is to find frequent itemsets (derived lists) of very large cardinality which can be in the order of thousands or millions, a number that corresponds to the number of iterations the *A-priori* algorithm would have to perform. Clearly, this process will not terminate in our setting. Instead we propose a novel adaptation that seeks to find "frequent" itemsets (derived lists) that are contained in the intersections of lists. In this new setting, the algorithm iterates over the number of intersecting lists and not their sizes. Thus, while *A-priori* in the k^{th} iteration finds frequent itemsets of size k, our adaptation finds frequent itemsets produced by intersecting k lists. Furthermore, unlike the original *A-priori* we are not given any monotonicity guarantees, that is if the intersection of k lists qualifies for use in the compression of the index then we cannot say that any subset of them also qualifies. The reason behind this is that in contrary to the *support* of an itemset which is just a count of many times the itemset exists in the dataset, the support (or gain in our terminology) of a derived list is a function of two variables (number of lists, number of items in the intersection of these lists), one increasing and one decreasing, while the algorithm augments the existing candidates. We discuss the adaptation of the *A-priori* algorithm in order to address these shortcomings in Section 3.

Conflicting Derived Lists: In Section 2 we explain that the candidate derived lists from a dataset cannot all be used for compression at the same time due to conflicts among them. Briefly, a conflict is present among a pair of derived lists when they have been constructed by using at least one common list and they also include at least one common item. Then using both of these lists for the compression of the index would result in duplicate items. The challenge is to select a subset of the candidate derived lists that presents no conflict and at the same time maximizes the compression ratio. In order to address this challenge we propose a heuristic based on an Integer Linear Programming modelling of the problem, which is presented in Section 4.

2 The Derived List

Given a set of lists $\mathcal{L} = \{l_1, l_2, \ldots, l_m\}$, a derived list dl is constructed by taking the items in the intersection of a subset $\mathcal{L}'$ of $\mathcal{L}$. We refer to the set $\mathcal{L}'$ as the base of dl and denote it as $dl.base$. We call $dl.items$ the set of items in the intersection of the lists in $dl.base$, and we refer to the number of items as $dl.size$. Last, we define a metric $dl.gain$, which is the gain (reduction in number of items to store) that the use of this derived list adds to the compression of the index. Assume a derived list dl and it's n base lists $l_1, l_2, \ldots, l_n$, where $n \geq 2$ ($n = |dl.base|$). If we select to materialize dl (e.g. write it to disk), then we can use it in each of it's base lists in the place of the original items, substituting them by a reference to dl. That essentially means that we are using only one copy of these items instead of n copies. Thus, when used in all of it's base lists, the gain of the dl (reduction of index size) is:

$$dl.gain = (|dl.base| - 1) \times dl.size \tag{1}$$

In order to compute the gain of a derived list we need to know the number of its base lists and the number of items in their intersection. Due to the way that we construct the derived lists, we know beforehand the number of base lists. That means that we just need to estimate the size of their intersection so as to estimate the derived list's gain. As we show next this can be done by calculating the Jaccard similarity of the lists using their already computed minhash signatures and a result based on the inclusion-exclusion principle.

Jaccard Similarity: Jaccard similarity is a measure of sets similarity, defined as the ratio of the size of their intersection over the size of their union. Given the sets $A_1, A_2, \ldots, A_n$, we compute their Jaccard similarity $JS(A_1, A_2, \ldots, A_n)$ as:

$$JS(A_1, A_2, \ldots, A_n) = \frac{|A_1 \cap A_2 \cap \ldots \cap A_n|}{|A_1 \cup A_2 \cup \ldots \cup A_n|} \tag{2}$$

Minhashing: Given a large collection of sets with values from a domain, minhashing [4] is the process of constructing a small signature for each one of them by applying a series of hash functions to each of the set elements. The minhash signatures present a very interesting property: given two sets A and B and their respective minhash signatures $mh(A)$ and $mh(B)$, the probability that $mh(A) = mh(B)$ equals to the Jaccard similarity between A and B.

Inclusion-Exclusion Principle: From the inclusion-exclusion principle [5] we know that we can calculate the union of a number of finite sets $A_1, \ldots, A_n$ as

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{k=1}^{n}(-1)^{k+1}\left(\sum_{1\leq i_1<\ldots<i_k\leq n}|A_{i1}\cap\ldots\cap A_{ik}|\right) \tag{3}$$

Size Estimation of Set Expressions: Assume n finite sets $A_1, \ldots, A_n$. We want to estimate the size of their intersection $|A_1 \cap A_2 \cap \ldots \cap A_n|$, given their respective minhashes $mh(A_1), \ldots, mh(A_n)$ and sizes $|A_1|, \ldots, |A_n|$. Let $S(A_1, \ldots, A_n)$ be the Jaccard similarity of sets $A_1, \ldots, A_n$:

$$S(A_1,\ldots,A_n) = \frac{|A_1\cap A_2\cap\ldots\cap A_n|}{|A_1\cup A_2\cup\ldots\cup A_n|} \tag{4}$$

The Jaccard similarity can also be estimated using the set's minhashes with strong guarantees as:

$$S(A_1,\ldots,A_n) = \frac{\boldsymbol{mh(A_1)}\cap\boldsymbol{mh(A_2)}\cap\ldots\cap\boldsymbol{mh(A_n)}}{\boldsymbol{mh(A_1)}\cup\boldsymbol{mh(A_2)}\cup\ldots\cup\boldsymbol{mh(A_n)}} \tag{5}$$

We continue by using the inclusion-exclusion principle (Equation 3) so as to recursively compute the intersection of the given sets. For $n = 2 * m$, the union of the n sets can be written as $U^- - a$, where U^- is the size of the union of the n sets minus the last term of the sum:

$$U^- = \sum_{k=1}^{n-1}(-1)^{k+1}\left(\sum_{1\leq i_1<\ldots<i_k\leq n-1}|A_{i1}\cap\ldots\cap A_{ik}|\right) \tag{6}$$

This last term, which is not present in U^-, is represented by a and is actually the size of the intersection of the n sets ($a = |A_1 \cap A_2 \cap \ldots \cap A_n|$). Then, the Jaccard similarity of the n sets can be expressed as:

$$S(A_1,\ldots,A_n) = \frac{a}{U^- - a} \tag{7}$$

and, thus, the intersection a is derived from the following formula:

$$a = \frac{S(A_1,\ldots,A_n)}{1+S(A_1,\ldots,A_n)} \times U^- \tag{8}$$

In the case that $n = 2 * m + 1$, the respective formula is:

$$a = \frac{S(A_1,\ldots,A_n)}{1-S(A_1,\ldots,A_n)} \times U^- \tag{9}$$

We can easily compute the Jaccard Similarity S for any combination of sets from their minhashes using Equation 5. We compute U^- recursively using Equation 6 and estimate the intersection size of the sets using Equation 8 (or 9).

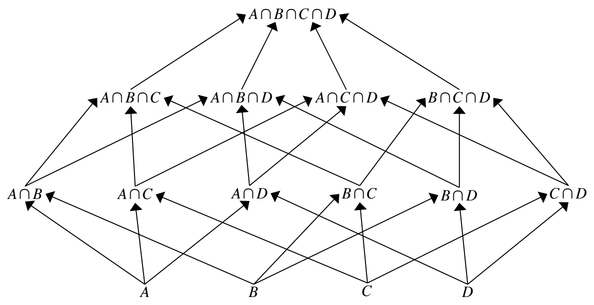


Fig. 1. The Derived Lists Lattice for lists A, B, C, D

Conflicting Derived Lists: In Figure 1 we depict the derived lists lattice that can be constructed from four example base lists A, B, C and D. The derived lists are connected with directed edges that form paths from the lower level (trivial derived lists with one base list) to the higher level (a single derived list constructed by intersecting all base lists). When there is a path connecting two derived lists these derived lists conflict, meaning that the use of one of them for compression suggests that we cannot use the other. This is explained from the fact that since they are on the same path they share at least one base list and also that the lower level one contains all of the items in the higher level one since the latter is constructed by further intersecting the former. However, this is not the only case where two derived lists conflict. Any pair of derived lists that share at least a base list and at least one item also conflicts. The important distinction between the two cases is that in the first case we can infer that two derived lists conflict without computing their intersection, while in the second case we have to compute their intersection in order to decide whether they conflict. We collectively refer to all the conflicting derived lists of a derived list dl by $dl.relatives$.

3 Candidates Generation

Candidates generation aims at discovering a set of candidate derived lists. Given a set of lists $\mathcal{L}$, at the end of the candidate generation phase we have populated a set of candidate derived lists $\mathcal{DL}$ described not by the actual items they include but by their base lists and their estimated gain.

Preprocess - Minhashing - LSH: The input data is the set of lists $\mathcal{L}$ in the inverted index. For each $l \in \mathcal{L}$ we compute it's minhash signature, and count it's size (number of items in l). At the same time we apply the LSH technique for minhashes [6] so as to populate buckets of similar lists. The buckets are populated with triplets of the form $(l.id, l.minhash, l.size)$. All these tasks are parallelizable and can be efficiently computed for large datasets using modern distributed processing frameworks such as Spark or MapReduce.

Generating the Candidate Derived Lists: For a derived list to qualify we demand that it's computed gain is greater than or equal to a user-defined *mingain* value. The default *mingain* value is 1, meaning that we add to the candidates set any derived list that offers even the minimum gain possible. We model our candidate generation problem as an intuitive variant of a frequent itemset mining problem, where we seek to find "itemsets" (derived lists) by intersecting base lists as we move upwards in the lattice. In this setting we would like to find itemsets (derived lists) with support (gain) that exceeds the value of mingain. While this adaptation is intuitive, there is a specific complication in our setting. The gain function is not a monotonous function because, while "moving" from the lower levels of the lattice to higher ones, it is computed as the product of an increasing function ($|dl.base| - 1$) by a decreasing one ($dl.size$), as intersecting more lists results in a new derived list of smaller or equal size. What this actually means is that the A-priori principle does not hold in our case. In order to overcome this problem, we are using a user-defined upper bound in the number of base lists that a derived list can have which we call k_{max}. That way we set a bound to the first term of Equation 1 (number of base lists). Parameters k_{max} and *mingain* can be tuned so that the system will favor either speed of execution, or compression ratio. Then, we solve Equation 1 for the second term, which is the size of the derived list. It follows that for a qualifying derived list dl of k^{th} order (having k base lists) it must hold that:

$$dl.size \geq \frac{mingain}{k-1} \tag{10}$$

Then, for $k = k_{max}$ it must hold that:

$$dl.size \geq \frac{mingain}{k_{max}-1} = minsize \tag{11}$$

where *minsize* is the minimum size a derived list can be of so as to benefit us with gain larger than *mingain*. This conclusion is justified by the fact that the size of a derived list dl_1 that is the intersection of $n \geq 2$ lists is less or equal to the size of a derived list dl_2 that is the intersection of any combination of m of those lists ($1 \leq m < n$). With this modification, our candidate generation problem reduces to the problem of frequent itemset mining and can be solved using e.g. the A-priori algorithm within each LSH bucket. An important difference that we must point out is that *minsize* is the minimum size for a derived list of order k_{max} to qualify, but since k_{max} is the highest order we use it as an overall minimum gain. The reason we use *minsize* for all orders in a first pass is that a derived list always reduces in size (or stays the same) when adding a new base list. In a second pass over the set of produced candidates within the bucket, we calculate the actual minimum size for each order and keep only the truly qualifying lists (e.g. those with $dl.gain \geq mingain$). As has been discussed, an important property of our technique is that all calculations are performed on the compact statistics that we maintain for each derived list, without looking at the original dataset. An optimization that we apply during candidate generation is the early pruning of candidates within a bucket by removing any of them that can be inferred to be a relative of another one with higher gain.

4 Derived Lists Selection

The input to this stage is the set of candidate derived lists generated in the previous phase. Each candidate derived list dl is described as a tuple ($dl.base$,$dl.gain$) where $dl.gain$ is an estimation of the gain as computed previously. Our goal is to select a non-conflicting subset of the candidate derived lists which maximizes the gain and then use it to compress the index. A natural choice for this kind of problems (selection-maximization) is ILP (Integer Linear Programming) [7].

ILP Formulation of the Problem: Let A be a matrix having as many rows as many conflicts exist among the candidates and one column for each candidate. For each conflict we add to A a row with the value 1 at the columns of the two conflicting candidates and 0 everywhere else. Also, let w be a vector of length equal to the number of candidates where $w[i]$ is the gain of the i^{th} derived list candidate. Finally, assume a solution vector x of length equal to the number of candidates with each position i taking a value of 1 or 0, respectively meaning that the i^{th} candidate is part or no of the solution represented by $\boldsymbol{x}$. We want to find the vector $\boldsymbol{x}$ that maximizes the gain $\boldsymbol{x} : max(\sum \boldsymbol{w} \cdot \boldsymbol{x})$ restricted by the rule $A \cdot \boldsymbol{x} \leq 1$ so that there are no conflicts in the solution. If we had the processing power required to solve the problem for all the candidates at once, it would suffice to feed A and w to an ILP solver and get the optimal solution in one step. In most practical cases, this would not be feasible when the number of candidates is in the order of thousands or millions. In such cases, we propose a greedy heuristic that uses the ILP approach, as described next.

Greedy Solver Algorithm: We load the candidate derived lists in decreasing order of their gains in a priority queue PQ, and we initialize an empty set $globalSolution$ to store the qualifying candidates. Then, depending on the ILP solver capabilities, we select a value N which is the number of candidates we shall use at each iteration. We get the top-N candidates out from the PQ, construct the corresponding matrix A and vector w, and pass them to the solver. The solution returned by the solver is a non-conflicting set of candidates that locally maximizes the gain for the candidates that it examined. We add this local solution to $globalSolution$ and remove from PQ any candidate that conflicts with any of the derived lists in the solution. Then, we start over using the next top-N candidates until PQ is empty or a desired compression ratio has been achieved.

5 Experimental Evaluation

We implemented our method using Apache Spark, Hadoop and HBase running on a small test cluster consisting of nine Linux virtual machines. The data used in our experiments were synthetically generated based on 20 publicly available real supply chain networks selected from [8]. Each supply chain is a directed acyclic graph and its nodes can be distinguished in source, inner and sink nodes. We created RFID data for each supply chain using a custom data generator, which loads the topology graph and the available average demand values for the terminal nodes, and then, using this information, performs

Table 1. Supply Chains Statistics

id	#nodes	#source nodes	#inner nodes	#sink nodes	#edges	#paths	max path length	#rfid records
1	40	12	26	2	48	22	8	7636760
2	152	21	33	98	211	1157	5	3037082
3	154	49	77	28	224	172	8	5122505
4	156	44	97	15	263	528	9	3956007
5	156	74	80	2	169	282	10	5132532
6	186	76	76	34	359	772	7	4224544
7	271	198	48	25	524	486	3	2533934
8	334	209	83	42	1245	4055	6	4119465
9	409	94	142	173	853	1158	3	2999721
10	468	401	65	2	605	579	6	2132407
11	482	418	52	12	941	889	5	3222313
12	577	398	89	90	2262	15181	8	3430016
13	617	128	124	365	753	3789	5	3490857
14	626	1	405	220	632	227	5	3553438
15	844	309	313	222	1685	2814	5	3283238
16	976	119	525	332	1009	1688	8	5799807
17	1206	1148	5	53	4063	4320	3	2001061
18	1386	619	731	36	1857	1140	6	5041132
19	1479	274	646	559	2069	6062	4	3998868
20	2025	820	646	559	16225	97085	4	3998940

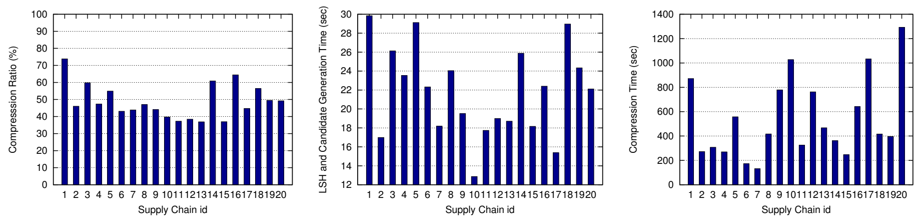
Table 2. Comparison of FIM algorithms and our Approx-A-priori

Number of Records		Time (ms)			
# graph walks	# rfid records	A-priori	FP-growth	Eclat	LSH+Candidate Generation
10	13	4	5	2	628
100	429	24	404,784	405,150	1,242
1,000	5,050	NA	NA	NA	1,253
10,000	51,270	NA	NA	NA	1,538
100,000	512,100	NA	NA	NA	3,867
1,000,000	5,122,505	NA	NA	NA	33,254

a number of random graph walks simulating tagged objects moving through the network. The generated walks are used to produce RFID records of format $(tagId, locationId, timestamp)$. Then, using these data, we create location-based indexes as inverted lists of the form $locationId \leftarrow list(tagId)$. Table 1 describes the supply chains and the dataset produced by performing one million graph walks on each of them. Because of the variable number of nodes and paths in each of the supply chains, the number of RFID records is not the same for all of them, as is depicted in the table.

Comparison with Frequent Itemset Mining (FIM) Algorithms: Our technique is aiming at the efficient discovery of common itemsets in a collection of lists. The most well known algorithms for such tasks are A-priori [3], FP-growth [9] and Eclat [10]. While these algorithms focus on the discovery of frequent itemsets in a large number of relatively small transactions, we are instead mostly interested in discovering large itemsets which are not necessarily very frequent. These algorithms are not optimized for computing very large itemsets that often arise by the intersections of very long lists. We used the implementations of A-priori, FP-growth and Eclat from [11] to demonstrate their inefficiency in such a setting, and the results are depicted in Table 2 (NA means that the respective algorithm either exited due to insufficient memory or space, or that it did not finish after running for twenty four hours). For fairness, all algorithms were executed in a single machine. Our method, by utilizing approximation and our novel adaptation of the A-priori algorithm within each LSH bucket, manages to scale to datasets orders of magnitute larger than the other algorithms do.

Compression Ratio Achieved: The compression ratio indicates the reduction in the size of the index due to the use of our techniques. Let $\mathcal{DL}$ be the set of the



(a) Compression Ratio (b) LSH/Cand.Gen. Time (c) Compression Time

Fig. 2. Compression Ratio and Execution Time for 20 supply chain networks

selected derived lists and $\mathcal{L}$ the lists in the original index. We define compression ratio as $\sum_{dl \in \mathcal{DL}} dl.gain / \sum_{list \in \mathcal{L}} |list.items|$. For these experiments we set k_{max}=5 and $mingain$=1. The value of the compression ratio achieved in each supply chain is presented in Figure 2(a). We observe an average compression ratio of 49% and we can see compression ratios well above 50% (maximum is 74%).

Execution Time: We measured the execution time for the three discrete stages of the method (LSH, candidate generation, compression). In Figure 2(b) we present the execution times for the LSH and the candidate generations stages. The time required for LSH is related to the number of RFID records generated for each supply chain, while the time required for the candidate generation phase is proportional to the number of total nodes contained in each supply chain. The execution times for running *Greedy Solver* to compress the indexes are depicted in Figure 2(c). LSH and candidate generation are parallelizable tasks and this results in fast execution times. Selection of derived lists and compression of the index is partly parallelizable and also performs heavy read operations while retrieving the original lists, so these operations are slower. We observe that the time needed to perform the compression of the index is proportional to the number of paths that exist in the network. A higher number of paths results in the creation of a larger number of candidates with more complicated relations.

Effect of parameters k_{max} and mingain: In order to examine how different values of k_{max} affect performance and outcome we ran a new set of experiments. We selected four out of the twenty supply chains and varied k_{max} from 2 to 7 ($mingain$=1). We observed that lower values of k_{max} are not desired since in some cases result in lower compression ratio with no gain in execution time. For higher values of k_{max} the required time decreases or remains on the same levels while someone would instead expect an increase. This behavior is explained by the early pruning optimization discussed at the end of Section 3. Next we varied $mingain$ value from 1 to 2000, keeping $k_{max} = 5$. As $mingain$ increased, the fewer the generated candidates were. For most of the datasets, as $mingain$ increased, the overall execution time decreased, but compression ratio remained at levels near the ones achieved by a value of 1. This is a consequence of the fact that the derived lists selected by *Greedy Solver* are the ones with the more gain and, thus, the ones with small gains are not likely to be selected anyway.

6 Conclusions

In this paper we presented a framework for compressing large inverted indexes built on low cardinality domains. Such indexes are common in many applications of interest and tend to produce lists that share large sets of common item references. Our techniques are able to discover the most promising of these common sets in a single pass over the original lists by utilizing novel dimensionality reduction and approximation techniques. We complemented our method with a greedy heuristic that uses an intuitive ILP formulation in order to select a subset of these common sets so as to construct a solution that maximizes the compression ratio. Finally, we implemented our framework using modern big data tools and used it for compressing data generated on real supply chain networks. As future work, we plan to explore ways to adopt our method so as to support incremental updates of the compressed indexes on streaming data.

References

1. Bleco, D., Kotidis, Y.: Rfid data aggregation. In: Proceedings of the 3rd International Conference on GeoSensor Networks. GSN '09, Berlin, Heidelberg, Springer-Verlag (2009) 87–101
2. Bleco, D., Kotidis, Y.: Business intelligence on complex graph data. In: Proceedings of the 2012 Joint EDBT/ICDT Workshops. EDBT-ICDT '12, New York, NY, USA, ACM (2012) 13–20
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of the 20th International Conference on Very Large Data Bases. VLDB '94, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1994) 487–499
4. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. J. Comput. Syst. Sci. **60**(3) (June 2000) 630–659
5. Knuth, D.E.: The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1997)
6. Rajaraman, A., Ullman, J.D.: Mining of massive datasets. Cambridge University Press, Cambridge (2012)
7. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1982)
8. Willems, S.P.: Data Set—Real-World Multiechelon Supply Chains Used for Inventory Optimization. Manufacturing & Service Operations Management **10**(1) (January 2008) 19–23
9. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. SIGMOD '00, New York, NY, USA, ACM (2000) 1–12
10. Zaki, M.J.: Scalable algorithms for association mining. IEEE Trans. on Knowl. and Data Eng. **12**(3) (May 2000) 372–390
11. Viger, P.F., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.W., Tseng, V.S.: SPMF: A Java Open-Source Pattern Mining Library. Journal of Machine Learning Research **15** (2014) 3389–3393