Knowl Inf Syst (2017) 50:969–997
DOI 10.1007/s10115-016-0962-8

REGULAR PAPER

SILVERBACK+: scalable association mining via fast list intersection for columnar social data

Yusheng Xie[1,3] · Zhengzhang Chen[2] · Diana Palsetia[1] · Goce Trajcevski[1] · Ankit Agrawal[1] · Alok Choudhary[1]

Received: 22 December 2014 / Revised: 14 April 2016 / Accepted: 27 May 2016 /
Published online: 4 July 2016

Abstract We present SILVERBACK+, a scalable probabilistic framework for accurate association rule and frequent item-set mining of large-scale social behavioral data. SILVERBACK+ tackles the problem of efficient storage utilization and management via: (1) probabilistic columnar infrastructure and (2) using Bloom filters and sampling techniques. In addition, probabilistic pruning techniques based on Apriori method are developed, for accelerating the mining of frequent item-sets. The proposed target-driven techniques yield a significant reduction of the size of the frequent item-set candidates, as well as the required number of repetitive membership checks through a novel list intersection algorithm. Extensive experimental evaluations demonstrate the benefits of this context-aware consideration and incorporation of the infrastructure limitations when utilizing the corresponding research techniques. When compared to the traditional Hadoop-based approach for improving scalability by straightforwardly adding more hosts, SILVERBACK+ exhibits a much better runtime performance, with negligible loss of accuracy.

Keywords Association rule mining · Frequent item-set mining · Columnar probabilistic databases · Social media · Bloom filter

1 Introduction and motivation

In order to increase the revenue via effective advertising, economic analysts have proposed a range of techniques in the recent years, collectively named *behavioral targeting* [7]. In the context of social websites, users' behavioral data may be generated via forms of likes, posts, retweets or comments [30,31]. However, regardless of the particular manner in which the

✉ Yusheng Xie
yxi389@eecs.northwestern.edu

1 Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL, USA

2 NEC Laboratories America, Princeton, NJ, USA

3 Baidu Research, Sunnyvale, CA, USA

data is generated, the foremost characterization of social behavior data is the large number of users got involved. For example, throughout our earlier projects [32] we have estimated that in March 2012 nearly 1 billion of public comments or post likes were generated by Facebook users alone.

Researchers often rely on the data mining techniques (developed for more traditional data sources and formats) to extract valuable knowledge from behavioral data. It is not surprising that analyzing the public social web and extracting the most relevant items (i.e., frequent item-sets) is a valuable application of association rule mining to large behavioral databases for a particular commercial interest. Broadly speaking, an *interest* could mean a group of online users, a brand or a product—e.g., the brand "Nikon" is a description of an interest in cameras. Given a set of interests and a large behavioral database of transactions of user activities in online social networks, one of the challenging tasks is to find a list of relevant interests that share a similar demographic characteristics. In many regards, this operation is analogous to finding frequent item-sets and association rules from a large number of transactions of co-occurrences of the items [1].

When it comes to online behavioral settings, the *scale* of the data is one factor posing unique challenges of a nature different from the contexts of the existing works on association mining. We are challenged with a behavioral database containing over 10 billion transactions, up to 30,000 distinct items and growing by over 30 million transactions every day. In this spirit, there are two important and, in some sense, complementary observations which motivated our work:

1. Adding more hardware could readily help in addressing the scalability problem—however, what if a Big Data startup cannot afford this "brute force" avenue of attaining sufficient computing power? Contrary to large enterprises such as Facebook or Twitter, many of their smaller-in-scale partner startups have much fewer database engineers. Those engineers are challenged with designing a system, which is expected to handle inundating amount of data sent from their larger social network partners. Constraints, both in terms of the budget limitations and considerations for energy-saving, dictate the necessity for designing alternatives on commodity hardware, as opposed to the simplistic "put it on more machines and scale".
2. There is the ever-present trade-off between the amount of storage use and the quality/utility of the knowledge obtained. Various forms of data compression techniques, both in the context of traditional files [24] as well as streaming, spatiotemporal and sensor data [5,10,16] have been proposed in the research literature. We note, however, that properly exploiting statistical techniques can shift the trade-off margin toward providing a significant increase in the utility of extracted knowledge from the large-scale behavioral data, at the smaller-scale storage overheads. Probabilistic approaches, for as long as they provide certain accuracy guarantees of the mining results, do seem like viable avenues toward efficient storage schemes.

1.1 Problem description

The main challenge in our application scenarios is to parsimoniously and accurately compute target-driven frequent item-sets and association rules for a given (large) database of users' activity logs, for the purpose of providing a real-time on-demand response.

We use $\mathcal{D}$ to denote the list of users' activities across public walls in the Facebook network (or handles from Twitter). $\mathcal{D}$ consists of quadruples of the form $(u_i, w_i, t_i, a_i) \in \mathcal{D}(i = 0, 1, \ldots, |\mathcal{D}|)$, each quadrupl denoting an individual user activity. The interpretation is that for ith transaction, user u_i made activity of type a_i on wall w_i at timestamp t_i. Each u_i

belongs to U, the set of all user IDs; each w_i belongs to W, the set of all wall IDs. In practice, $|W| \ll |U| \ll |\mathcal{D}|$, thereby justifying the expectation that for some $i \neq j$, $u_i = u_j$ or $w_i = w_j$.

Aggregating the wall IDs in transactions from $\mathcal{D}$ by user ID generates $\mathcal{D}_U$—which is a database of *behavioral* transactions. There is a clear analogy between $\mathcal{D}_U$ and the famous supermarket example of frequent item-set mining. User IDs in $\mathcal{D}_U$ are equivalent to transactions of purchase; walls that a particular user has activities upon are equivalent to the items purchased in a particular transaction. In this paper, we use *wall* and *item* interchangeably.

For a given (minimal) support level α, a frequent item-set F is a subset of W such that there are at least α transactions in $\mathcal{D}_U$. F_k, a k-item-set, denotes a frequent item-set with exactly k number of items. A target-driven *rule* is generally defined as an implication of the form $X \Rightarrow Y$, where $X, Y \subset W$, $X \cap Y = \emptyset$, $X \cup Y = F_k$ and Y is given as the target.

Given a live and rapidly growing $\mathcal{D}$ and a target Y, our goal is to efficiently discover rules that imply Y. As an illustration, $\mathcal{D}_U$ in our settings is equivalent to an 800-million-by-30,000 table that would have over 20 *trillion* cells in full representation.

Our main contributions can be summarized as follows:

- We present SILVERBACK+—a probabilistic framework for accurate association rule and frequent item-set mining at massive streaming scale, implemented on a commodity hardware. The framework and algorithmic implementations have been successfully deployed at large scale for commercial use and progressively improved to the current version since May 2011.
- We present algorithms based on hierarchy of Bloom filters and sampling techniques which yield fast probabilistic query processing with satisfactory penalties on the accuracy, using column-based storage for managing large transactional databases.
- We propose an Apriori-based algorithm to probabilistically prune candidates without support-counting for every candidate item-set.
- We present quantitative observations based on a large set of experiments that we conducted, demonstrating that SILVERBACK+ is significantly more efficient than a generic MapReduce-based implementation.

A preliminary version of this work was presented in [32], and the current article, in addition to the more comprehensive set of experiments, improves the earlier work by introducing hierarchical (binary) tree of Bloom filters and corresponding algorithms for speeding up the computation of the two frequent item-sets and association rules.

In the rest of this article, Sect. 2 presents the related literature and observations regarding our proposed approach. Sections 3 and 4 address in greater detail the storage and infrastructure, along with the methodology and data structures and algorithms. In Sect. 5, we present the experimental results, and in Sect. 6, we conclude the article and outline directions for future work.

2 Related work

We now present an overview of the relevant literatures. After discussing three categories of works related to association mining, we overview the applications of Bloom filters.

2.1 Association mining

Association mining focuses on efficient detection of correlations between items in a dataset. Despite several recent advances in parallel association mining algorithms [22,33], the core

techniques are still similar in spirit to the popular Apriori algorithm [2]. Essentially, Apriori identifies the frequent items by starting with a collection of small item-sets and proceeding to larger item-sets only when all the subsets happen to be frequent. This incurs cost-overheads due to scanning the entire database in every count step. Several techniques have been proposed to improve issues of Apriori such as counting step, scanning and representing database, generating and pruning candidates and ordering of items, some of which we discuss in detail as follows.

2.1.1 Max-miner

Max-Miner [3] addresses the limitations of basic Apriori by allowing only *maximal frequent item-set* (long patterns) to be mined. An item-set is maximal frequent if it has no superset that is frequent. This reduces the search space by pruning not only on subset infrequency but also on superset infrequency.

Max-Miner uses a set enumeration tree which imposes a particular order on the parent and child nodes, but not its completeness. Each node in the set enumeration tree is considered as a candidate group (g). A candidate group consists of 2-item-sets: first called *head* ($h(g)$), which is the item-set enumerated by the node, and the second called *tail* ($t(g)$), which is an ordered set and contains all items not in $h(g)$. The ordering in the tail item-set indicates how the subnodes are expanded. The counting of support of a candidate group requires computing the support of item-sets $h(g)$, $h(g) \cup t(g)$, $h(g) \cup \{i\}, \forall i \in t(g)$. Superset pruning occurs when $h(g) \cup t(g)$ is frequent. This implies that item-set enumerated by subnode will also be frequent but not maximal, and therefore, the subnode expansion can be halted. If $h(g) \cup \{i\}$ is infrequent then any head of a subnode that contains item i is infrequent. Consequently, subset pruning can be implemented by removing any such tail item from candidate group before expanding its subnodes.

Although Max-Miner with superset frequency pruning reduces the search time, it still needs many passes of the transactions to get all the long patterns—becoming inefficient in terms of both memory and processor usage (i.e., storing item-sets in a set and iterating through the item-sets in the set) when working with sets of candidate groups.

2.1.2 Divide and conquer approaches

FP-Growth [12] gains speed-up over Apriori by allowing frequent item-set discovery without candidate item-set generation. It builds a compact data structure called the FP-tree which can be constructed by allowing two passes over the dataset, and frequent item-sets are discovered by traversing through the FP-tree.

In the first pass, the algorithm scans the data and finds support for each item, allowing infrequent items to be discarded. The items are sorted in decreasing order of their support. The latter allows common prefixes to be shared during the construction of FP-tree. In the second pass, the FP-tree is constructed by reading each transaction. If nodes in the transaction do not exist in the tree, then the nodes are created with the path. Counts on the nodes are set to be 1. Transactions share common prefix item, and the frequent count of the node (i.e., prefix item) is incremented.

To extract the frequent item-sets, a bottom up approach is used (traversal from leaves to the root), adopting a divide and conquer approach where each prefix path subtree is processed recursively to extract the frequent item-sets and the solutions are then merged.

Allowing fewer scans of the database comes at the expense of building the FP-tree—the size of which may vary and may not fit in memory. Additionally, the support can only be computed once the entire dataset is added to FP-tree.

Similar to FP-Growth, Eclat employs the divide and conquer strategy to decompose the original search space [34]. It allows frequent item-set discovery via transaction list (tid-list) intersections and is the first algorithm to use column-based, rather than row-based representation of the data. The support of an item-set is determined by intersecting the transaction lists for two subsets, and the union of these two subsets constitutes an item-set.

The algorithm performs depth-first search on the search space. For each item, in the first step it scans the database to build a list of transactions containing that item. In the next step, it forms item-conditional database(if the item were to be removed) by intersecting tid-list of the item with tid-lists of all other items. Subsequently, the first step is applied on item-conditional database. The process is repeated for all other items as well.

Like FP-Growth, Eclat reduces the scans of the database at the expense of maintaining several long transaction lists in memory, even for small item-sets.

2.1.3 Distributed and parallel approaches

Discovering patterns from a large transaction dataset can be computationally expensive, and therefore, almost all existing large-scale association rule mining utilities are implemented on the MapReduce framework. Such examples include Parallel Eclat [35], Parallel Max-miner [8], Parallel FP-Growth [21] and Distributed Apriori [33].

Table 1 compares our proposed method with other popular existing methods in many aspects including their scalability to more nodes (SILVERBACK+'s core complexity analysis is presented in Sect. 4.3.3). For the number of database scans required by different algorithms, Apriori and Max-Miner would require $O(k)$ scans where k is the number of longest frequent pattern length. On the other hand, Eclat, FP-Growth and SILVERBACK+ would only need a constant time of database scans. For memory footprint, we try to provide some qualitative comparison in Table 1. Generally speaking, Apriori and Max-Miner require an amount of memory consistent with the database size (number of transaction) and the problem size (minimal support, maximal pattern length). FP-Growth is known to consume a large amount of memory due to the growth of its tree structure [21,23]. Eclat and SILVERBACK+ can achieve relatively low memory footprint by taking advantage of the columnar storage (see discussion on Figure 22 of [34]). And SILVERBACK+ further cuts its memory footprint thanks to the compressibility of the Bloom filters.

2.2 Modern applications of Bloom filters

Capturing demographics between any two interests can generate high space complexity as it requires membership operations. Bloom filter is a popular space-efficient probabilistic data structure used to test membership of an element [4]. For example, Google's Bigtable storage system uses Bloom filters to speed up queries, by avoiding disk accesses for rows or columns that do not exist [6]. Similar to Google's Bigtable, Apache modeled the HBase, which is a Hadoop database. HBase employs Bloom filters for two different use cases. One is to access patterns with a lot of misses during reads. The other is to speed up reads by cutting down internal lookups.

A nice property of Bloom filters is that the time needed to incorporate new items or to check whether a given item is in the set, is fixed at $O(k)$, where k is the number of hash functions, and is independent of the items already present in that set. However, there is a caveat: It

Table 1 Comparison with popular association mining algorithms

Algorithm	Transaction storage	Freq. itms. representation	Db. scans	Memory footprint	Cluster scalability	Empirical efficiency	Support count	Lines of code	Accuracy
Apriori	Row-based	Row-based	$O(k)$	Large	Good (Ye and Chiang [33])	Benchmark	Yes	∼1000	Exact
Max-Miner	Row-based	Row-based	$O(k)$	Large	Fair (Chung and Luo [8])	∼5×	Yes	Unknown	Exact
Eclat	Columnar	Flexible	Const.	Small	Poor (Zaki et al. [35])	3×–10×	Yes	∼2000	Exact
FP-Growth	Row-based	FP-tree	Const.	Enormous	Very good (Li et al. [21])	5×–10×	Yes	7000+	Exact
SILVERBACK+	Columnar	Flexible	Const.	Tiny	Good	>15×	Const. time	∼2000	Probabilistic

For exact mining, Eclat is often favored on small, fit-in-memory problems and parallel FP-Growth is sometimes favored for solving a large problem in a distributed setting. Our proposed SILVERBACK+ achieves superior in efficiency, scalability and memory footprint by adapting probabilistic data structures

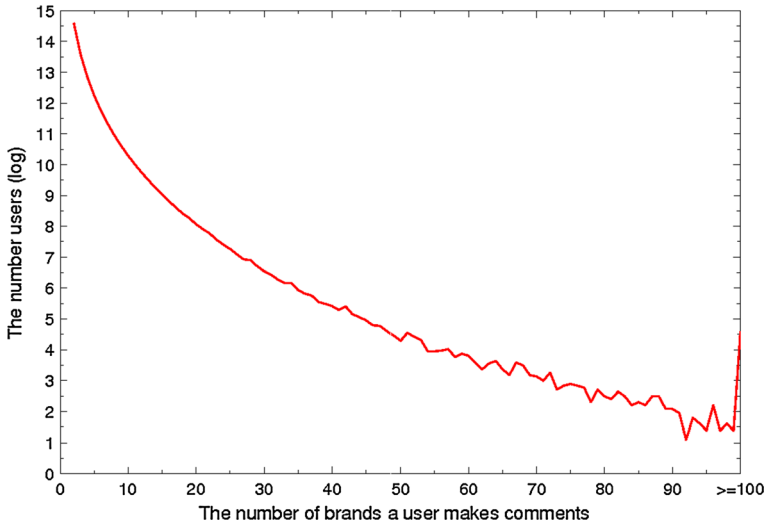


Fig. 1 Facebook user activity distribution (June 2008–January 2012, with *vertical axis* in log scale)

allows for false positives. For a given false positive probability p, the length of a Bloom filter m is proportionate to the number of elements n being filtered: $m = -n \ln p/(\ln 2)^2$.

A recent application of Bloom filters in association mining appears in [25]. The authors of [25] propose a sophisticated and interesting proposal to use Bloom filters to preserve privacy in association mining. The focus of [25] is primarily on the probabilistic and random nature of keyed Bloom filters for preserving privacy and is not on their spatial efficiency and scalability in a practical distributed environment (e.g., the authors of [25] test up to 515 K transactions on real data and 1 M transactions on synthetic data on a shared memory platform).

3 Storage and infrastructure

Given the scale of behavioral database $\mathcal{D}_U$ in our settings, the traditional row-based storage [2,12,19] would be a poor choice for scalability. To improve performance and scalability, we demand an efficient storage scheme. It is not our objective to invent a general-purpose advanced distributed storage engine, adding to the already abundant list of such engines and file systems. Instead, we focus on an application/data-driven ad hoc solution and, as it turned out, a probabilistic column storage is very effective in tackling the massive data scale in our intended application domains.

3.1 Column storage and scalability

The key motivation for our design is based on the following observations: (1) The full representation of $\mathcal{D}_U$ full requires over 20 trillion cells (740 M users by 32 K walls). This is impractical even in distributed environment, aside from budgetary issues. (2) Of the 20 trillion cells, less than 0.1 % are populated. According to our estimates, an average user accesses less than 14 of the 32 K walls.

The sparsity of $\mathcal{D}_U$ is not a coincidence, nor should it come as a surprise. In fact, the global sparseness in a social graph and the power law decay in its node degree distribution are examples of the asymptotic behaviors we observe. Figure 1 illustrates the distribution of

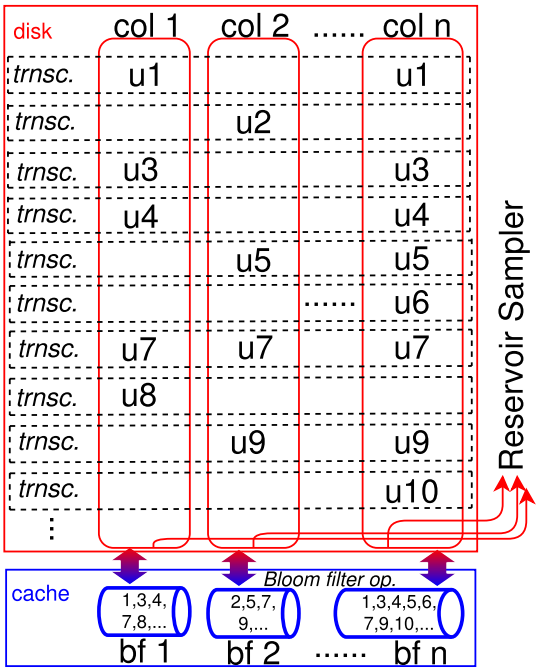


Fig. 2 Using columnar storage in place of traditional row-based storage for transactions (abbreviated as *trnsc.* in figure), with probabilistic enhancement. Each transaction is visually grouped by a *dashed rectangle* but is physically distributed among different columns (abbrv. as *col 1*, . . ., *col n*). Each *col* is compressed to a Bloom Filter (abbrv. as *bf 1*, . . ., *bf n*), which is stored in cache and is easier to access than the uncompressed columns. We use *red* (*upper rectangle*) to denote what is stored on large but slow hard disks and *blue* (*lower rectangle*) for small but faster cache storage (e.g., RAM or SSD). The bidirectional *vertical arrows* between *disk* and *cache* denote the insertion of columnar items to the Bloom filters (an algorithmic process that is further explained in Fig. 3). *Reservoir sampler* on the *right* suggests that sampling would occur in the insertion process if (and only if) some columns are prohibitively large for the *cache*. Best viewed in *color* (color figure online)

Facebook users and the number of walls (items) they access, demonstrating that the number of users accessing x number of walls drastically decreases as x increases. Specifically, over 40 % of the users only access less than 5 of the 32,000 walls—with a note that the "spike" on the right side is due to aggregating all users with more than 100 accessed walls into a single category. Hence, majority of the transactions in $\mathcal{D}_U$ are likely to contain only a small number of items.

We use a sparse representation of the massive $\mathcal{D}_U$ called "list of lists" (LIL) [26] (or "Column Family" in Cassandra [18]). LIL typically stores a massive sparse matrix by using a list to record the nonzero cells for each row. A column-based "list of columns" (LIC) representation is implemented for representing $\mathcal{D}_U$. That is, the LIC representation of $\mathcal{D}_U$ contains a wall column for each wall ID, and each wall column only contains the active user IDs of the 800 million users. The upper part of Fig. 2 illustrates how the traditional row-based transactions of items can be stored as columnar database, where LIC is a popular implementation.

One of the advantages in this columnar storage is data independence: the LIC representation of $\mathcal{D}_U$ can be partitioned by columns which can be stored as physically different files on different hosts. Inserts, deletes and updates to any wall will only affect its column and therefore avoids database locks, which is especially helpful for live databases such as $\mathcal{D}_U$.

3.2 Probabilistic enhancement

An important consequence of the sparsity of $\mathcal{D}_U$ is that in LIL representation, the lists/columns for the walls will have drastically different lengths. For example, the wall list for Coca Cola on Facebook contains over 30 million user IDs, whereas the small (albeit important) interests such as ACM SIGMOD have less than 100 user IDs in their lists.

The main problem caused by the massive size differences is that the resource allocator would face a combinatorial problem—each host has a capacity and each column has different sizes. If all the columns were similar in size, the allocator could have easier situation to deal with by treating all columns equally. Toward this, two approaches seem appealing:

1. One may opt to shard the longer columns (e.g., Coca Cola)—however, this introduces extra complexity as it diminishes the strong inter-column independence, which is important for us to scale easily. Extra locks would be required at column-level and shard-level for different chunks of a sharded column. The situation becomes more complicated if the column is so big that its shards reside on multiple hosts. Indeed, sharding functionality is available in existing products like MongoDB.[1] But MongoDB 2.1 generically implements readers–writer lock and allows one write queue per database, which is not desirable in our case and may have unforeseeable impact at large scale.
2. The other approach—which we adopted as our philosophy—is to simply solve the locking problem by "avoiding it". Namely, similar to [14], we impose each column file to be single-threaded, and therefore, no lock mechanism or extra complex management is required. The trade-off here is the need to make sure each column file size can be handled by a single thread with a reasonable delay. Sampling can alleviate the size difference among columns and make large columns controllable by a single-thread, and reservoir sampler [29] is used for exceedingly long columns. In practice, we sample 500,000 IDs for columns with more than 500,000 IDs. A bonus of using reservoir sampler is the ability to incrementally update the pool as new IDs are added to a given column and guarantee that the pool is a uniform sample of the entire column at any given moment. For each sampled column, an extra field is required to record the sampling rate.

Now the main problem is that the column files still cannot fit into the main memory of a modest cluster, even after sampling—e.g., loading all column files of the described $\mathcal{D}_U$ requires roughly 300 GB after the sampling. The practical goal is to reduce the representation of $\mathcal{D}_U$ from 300 GB down to approximately 25 GB—without breaking data independence, performance or scalability. With such constraints, our options are limited by "facts of life" such as: (a) sampling-based techniques cannot be used since any sampling would have happened in the previous stage; (b) coding-based information compression is also undesirable because of its impact on performance and updatability.

Given these observations, Bloom filter [4] with its probabilistic storage efficiency seems like a plausible choice. Given its space efficiency for probabilistic testing of set membership, our idea is to construct a Bloom filter for each column, as depicted in the bottom part in Fig. 2. When the Bloom filters are built, they are meant to be cached in memory while the much larger columns can reside on slower disks. In our experience, Bloom filters' efficiency is about 5 to 7 bits per ID, where each ID is originally stored as a string of 10 to 20 ASCII characters, depending on the chosen column. In addition to drastically reducing the storage size, Bloom filter files can be incrementally updated as more IDs are added to the corresponding column file, which means no rebuild is necessary for the filters.

[1] http://www.mongodb.org.

Although the Bloom filters created for different columns can use different number of hash functions, different false positive rate or different number of set bits, we need to make sure all Bloom filter arrays are of the same size. In practice, we enforce the Bloom filter size to be 7,000,000 bits $=$ 854.5 KBytes, which guarantees less than 0.1 % false positive rate with 500,000 expected inserts. Doing the same for all 30,000 columns would yield 854.5 KBytes $\times$ 30,000 $<$ 24.5 GBytes. That is, we expect at most 500,000 (the number of max sample size) IDs to be added to any Bloom filter. Assuming that each ID sets seven different bits in the filter, at most 50 % of the bits in the Bloom filter will be set which, in turn, guarantees the bound on the false positive rate on the filters.

Together, the sampling limit and the size of the filter guarantee an acceptable or satisfactory level of accuracy. While this equal-in-size requirement might seem unnecessary and even superfluous, it is specifically imposed to enable bit operations between any two Bloom filters, which is critical in our association mining algorithm. As our experiments have demonstrated, both the sampling and Bloom filter have a very limited impact on the accuracy of the results.

4 Algorithmic methodologies

The execution of popular algorithms such as Apriori [2] and FP-Growth [12], even their distributed implementations [33], is row-based, transaction row being the main execution unit. However, within the proposed storage scheme this assumption is no longer valid and it is not straightforward to apply the existing algorithms to accommodate to our storage, due to the fundamental differences in data scanning between row-wise storage and columnar storage.

In our methodology, mining frequent 2-item-sets is treated separately from frequent n-item-sets for $n > 2$ due to special characteristics in 2-item-sets that lead to a very fast algorithm. First, based on the proposed columnar probabilistic storage scheme, we describe a simple algorithm for finding frequent 2-item-sets that uses Apriori Principle (cf. Algorithm 1). Then, we propose a faster alternative algorithm for mining 2-item-sets, since frequent 2-item-sets are the prerequisite for all frequent n-item-sets for $n > 2$. This faster method, described in Algorithms 2, 3 and 4, relies on a novel hierarchical tree structure of Bloom filters that helps in minimizing membership checks against Bloom filters. Finally, in Algorithm 6, we develop an algorithm for mining n-item-sets, $n > 2$, which is made efficient by using a minHash-based probabilistic pruning technique, described in Algorithm 5.

4.1 Mining frequent 2-item-set with Apriori

We first demonstrate the column-oriented algorithm for finding frequent 2-item-sets $\{X = \{x\}, Y = \{y\}\}$, where X and Y are both single item-sets with a given minimal support α. The two item-set algorithm is often used in our practical usage, where the owner of a brand y is interested in finding out other brands that are most frequently associated with y.

All the possible candidates for x are elements from W, the set of all items. Algorithm 1 starts by filtering out the unqualified candidates whose support is below α—a process can be done very efficiently by scanning $O(|W| - 1)$ numbers, since the algorithm simply queries the length of each column file.

Let $W' \subseteq W$ denote the subset of W, which contains all the walls whose column size is above α. For each $y \in W'$, the algorithm loads the user IDs from column y into a set U_y. Since the actual user IDs are not explicitly stored with the Bloom filter and reside on a much

Algorithm 1: Column-oriented algorithm for finding frequent 2-item-sets and association rules

Input: α, minimal support, W, set of all items, $\mathcal{D}_U$, the database of transactions
Output: O, set of all frequent 2-item-sets

```
1  W' ← {x|x ∈ W, length of x column ≥ α}; O ← {}
2  for each y ∈ W' do
3  |  U_y ← IDs from y column
4  |  for each x ∈ W' and x ≻ y do
5  |  |  support_{x,y} ← 0
6  |  |  bf ← x column's Bloom filter
7  |  |  for each u ∈ U_y do
8  |  |  |  if u in bf then
9  |  |  |  |  support_{x,y} += 1
10 |  |  if support_{x,y} ≥ α then
11 |  |  |  append {x, y} to O
12 return O
```

slower disk, reading user IDs from disk only happens once per wall to avoid cost (note that U_y at each iteration is small enough to fit in memory). In other words, the algorithm scans the whole database from the disk only once. Then for each wall's Bloom filter representation b_x, where $x \in W'$, the algorithm tests whether u is a member of b_x for $\forall\, u \in U_y$. By testing U_y against b_x, the algorithm effectively finds (with false positives introduced by the use of Bloom filter) $y \cap x$, the intersection between y column and x column. At this stage, confidence and support filtering is applied and all qualified y columns are put into the output set O. The $x \succ y$ constraint says that x must come after y in atomic order, which guarantees that $\{x, y\}$ and $\{y, x\}$ are not calculated twice.

Algorithm 1, based on Apriori Principle, is intuitive to understand. However, it incurs a large number of membership checks in Bloom filters (which, being very fast with non-cryptographic hash calculation, is still the main cost of running Algorithm 1). In the following section, we propose an alternative algorithm, based on hierarchical binary tree of Bloom filters, to reduce such membership checks.

4.2 Mining frequent 2-item-set with hierarchical binary tree of Bloom filters

Frequent 2-item-sets are probably the most widely used and are the foundations for further calculation involving n-item-sets for $n > 2$. As a result, it is more desirable to expedite 2-item-set mining without introducing additional false negative results. In this section, we propose an alternative, faster algorithm that builds a hierarchical data structure on the Bloom filters and use this structure to speed up the mining of frequent 2-item-sets without introducing false negatives or any additional false positives. In addition, this algorithm can be easily extended for the general purpose of list intersection operations.

We use dictionary Z to denote all Bloom filters including the leaf ones from Algorithm 1. The number of items in Z is at most $2|W| - 1$ and equality only holds when items in Z is 2^n for some integer n. We use the dictionary notation, $Z[key] = value$, where "key" identifies the Bloom filter stored as "$value$". The filters, as shown in Algorithm 2, are organized into a binary tree. To store the information of how the binary tree is constructed (how the nodes are linked), we use a list H. Each item in H is a *(leftchild,rightchild,parent)* tuple. In addition, Algorithm 2 assumes that H must be sorted in the descending order of their distances to the root of the tree.

Algorithm 2: Build hierarchical binary tree H and the Bloom filters Z

Input: W, list of all items, M list of metrics corresponding to items in W.
Output: H, tuple-represented binary tree, Z, dictionary of Bloom filters for all nodes in H

```
1  Z ←empty dictionary
2  H ← linkage(M)
3  for tuple (leftchild,rightchild,parent) ∈ H do
4      for child ∈ [leftchild,rightchild] do
5          if child ∈ W then
6              Z[child] ← child column's Bloom filter
7          else
8              leftgrandchild ← H.findByParent[child].getLeftChild
9              rightgrandchild ← H.findByParent[child].getRightChild
10             for grandchild ∈ [leftgrandchild, rightgrandchild] do
11                 if grandchild ∈ W then
12                     Z[grandchild] ← grandchild column's Bloom filter
13             Z[child] ← Z[leftgrandchild] bitOR Z[rightgrandchild]
14 return Z
```

In Algorithm 2, the two data structures, Z and H, are built. In short, Algorithm 2 constructs a binary tree of Bloom filters, where every parent node is the logical or result of its two children filters. We use Z to denote this structure. And the leaf filters in the constructed tree correspond to items in W, so they are directly computed from the original database. The linkage() function in line 2 of Algorithm 2 is a pairwise hierarchical clustering algorithm that produces H. We use the metric information stored in M to compute a binary hierarchical cluster. Readers can assume that the ith item in M corresponds to the ith item in W. The items in M can be multi-dimensional. For example, $M = \{(\text{support}(i), \text{support}(i)^2) | i \in W\}$ is a simple choice of a two-dimensional M. The choice of M will have an impact on the practical performance of the resulting H, a claim we will empirically evaluate below in experiments. Figure 3 depicts a mini-example of the hierarchy coming out of Algorithm 2. More details about how to implement the linkage() function can be found in [13], which is based the MATLAB package and SciPy project.

The linkage output H is basically a binary tree (not necessarily a complete binary tree), whose leaf nodes correspond to the each row in M. Each non-leaf node in H is a merge point with a calculated distance based on its left subbranch and right subbranch. As a result, by choosing a target intra-cluster distance, getting a clustering on the items in M is simply making a cut below the non-leaf nodes whose left/right subbranches are above the target distance. By choosing varying target distances, one gets a hierarchical cluster.

Algorithm 3 illustrates how the basic frequent item-set mining algorithm (Algorithm 1) can benefit from consulting information in the linkage output H. By performing membership lookups from the top nodes to leaf nodes in H, the algorithm can avoid a lot of redundant checks in the subbranches by eliminating negative memberships from nodes closer to the root. Also we notice that this construction does not introduce any additional false positivity to the final results other than the false positivity built into the leaf-level filters.

Algorithm 4, used as a subroutine in Algorithm 3, is a core component that determines the temporal performance of the overall algorithm. It encodes the branching and pruning decision at each node in the binary tree structure Z built from Algorithm 2. The version described in Algorithm 4 would eagerly attempt to advance to subbranches at the minimal condition: When the number of positive Bloom filter tests is greater than α, the minimal supports threshold. The motivation for this aggressive advancing(therefore, conservative pruning) strategy is to minimize the impact from overly OR-ed Bloom filters at higher levels of the tree. Even

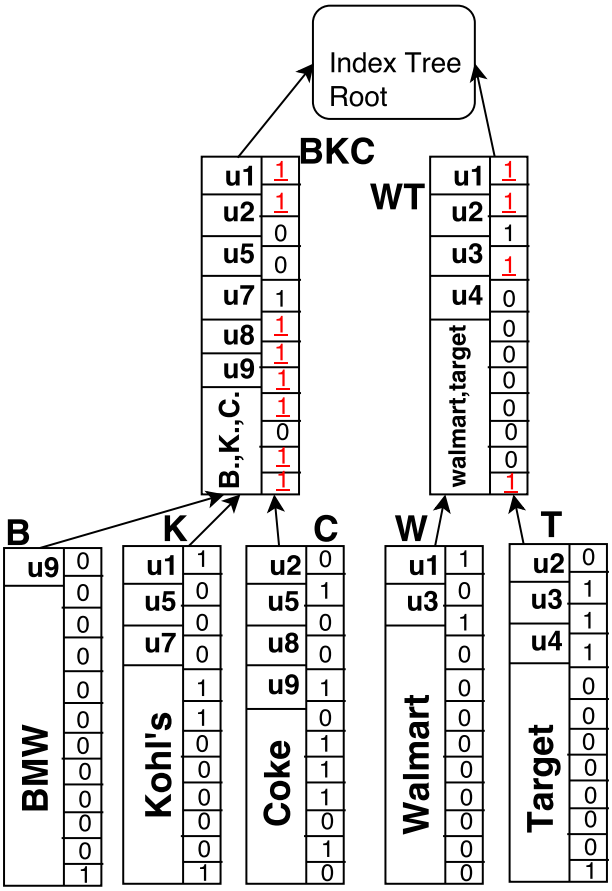


Fig. 3 Each tree node contains Bloom filter structure based on local lexicon. A bit is marked *red* and *underlined* if it is set during subtree merge. Best viewed in *color* (color figure online)

presented with a completely filled Bloom filter, Algorithm 4 can terminate and advance to its subbranches in $O(\alpha)$. In addition to $prospect$, the binary branching decision, Algorithm 4 also returns F, a set of IDs that have failed the Bloom filter test. The purpose of maintaining F is to exclude all elements in F from future Bloom tests in all subbranches under that node. The implication is that any ID will be only tested for negative at most once in the *entire* tree and therefore can vastly reduce repetitive membership checks. The fast_prune algorithm, as shown in Algorithm 4, trades off possibly greater speed-up for much better worst-case performance. It is entirely reasonable and possibly profitable to change to a less conservative pruning strategy. Simple heuristics such as "**if** $count \geq \alpha$ **and** $count \geq |F|$" can apply to line 7 of Algorithm 4. This heuristic effect is to consider the "quality" of the current Bloom filter. If the filter is close to being full, $count$ will quickly grow larger than $|F|$ and drive the algorithm to stop pruning and advance to subbranches. This logic is profitable because when a Bloom filter gives a higher percentage of positives, the filter is likely to be over populated. On the other hand, if the filter is sparse, $|F|$ will grow slowly and this condition in line 8 will not be met until a lot of IDs are pruned out.

Algorithm 3: Improved column-oriented algorithm (from Algorithm 1) for finding two frequent item-sets and association rules

Input: α, minimal support, W, set of all items, Z, dictionary-represented binary tree of Bloom filters,H, tuple-represented binary tree.
Output: O, set of all frequent 2-item-sets

```
W' ← {x|x ∈ W, length of x column ≥ α}; O ← {}
for each y ∈ W' do
    U_y ← IDs from y column
    S ← empty stack;N ← empty dictionary
    S.push(H.root)
    while S not empty do
        n ← S.pop
        U_{y−n} = {x ∈ U_y|x ∉ N[n]}
        bf ← n column's Bloom filter
        if n ∈ W' then
            support ← 0
            for each u ∈ U_y do
                if u in bf then
                    support+ = 1
            if support ≥ α then
                append {y, n} to O
        else
            prospect, F ← fast_prune(U_{y−n}, bf)
            if not prospect then
                continue
            leftchild ← H.findByParent[n].getLeftChild
            rightchild ← H.findByParent[n].getRightChild
            S.push(leftchild, rightchild)
            N[leftchild] ← F ∪ N[n]
            N[rightchild] ← F ∪ N[n]
return O
```

Algorithm 4: Fast_prune algorithm

Input: U_y, list of user IDs from y column, bf a given Bloom filter,α, minimal support.
Output: $prospect$, a Boolean decision, F, a subset of U_y that failed the Bloom filter test with bf.

```
F ←empty set; count ← 0
for u ∈ U_y do
    if u∈ bf then
        count+ = 1
    else
        F.add(u)
    if count ≥ α then
        break
prospect ← (count ≥ α)
return prospect,F
```

4.3 Two issues with mining frequent n-item-sets ($n > 2$)

Two particular operations in the Apriori algorithm significantly slow down its execution time when mining n-item-sets ($n > 2$). The first is the multiple scans of transactions. The other operation that significantly contributed to the temporal cost of traditional Apriori is candidate pruning, which requires counting support for each candidate generated. To overcome

those two drawbacks, various pruning and optimization techniques have been proposed, as discussed in the related work section.

4.3.1 Minimizing scans of transactions

Apriori algorithm classifies candidate item-sets and explores their candidacy by the cardinality of the item-set, where at each cardinality level, the algorithm scans $\mathcal{D}_U$ (the entire database of transactions) for counting the supports of the candidate sets at that cardinality level. The problem then becomes obvious: The entire execution of the algorithm scans the database multiple times, which is not desirable.

Minimizing the iterations of scanning the database is critical in improving the overall efficiency of association mining algorithms, especially for large databases. FP-Growth [12] offers improvements partially due to the fact that it only scans the database of transactions twice in building the FP-tree structure. However, the size of the FP-tree structure can be large and reading frequent patterns from the FP-tree requires traversing through the tree which, in turn, still incurs multiple loads. Benefiting from its columnar storage, Eclat [34] reads activities/transactions column by column and only the necessary columns and intersections of columns are retrieved into memory when checking the candidacy of each candidate. Similar to Eclat, our proposition only retrieves the necessary column files each time and further minimizes the I/O by replacing intersections of columns by AND-masked Bloom filters.

4.3.2 Probabilistic candidates pruning

Traditionally, avoiding the exponential growth of candidate item-sets ($2^{|W|}$ possible candidates) by the Apriori Principle and other algorithmic improvements [3] is based on pruning the unqualified candidate item-sets. Apriori Principle becomes especially effective when

Algorithm 5: Apriori-gen algorithm for generating and probabilistically pruning candidates

Input: F_{k-1}, frequent $(k-1)$ item-sets; α, minimal support; $H_1(c), \ldots, H_f(c)$, *sorted* lists that holds the Bloom hash indices for $\forall c \in F_{k-1}$; S_c for $\forall c \in F_{k-1}$, support counts for all frequent $(k-1)$ item-sets

Output: C_k, set of candidates for frequent k item-sets after pruning

$C_k \leftarrow \{\}$
for $c_1, c_2 \in F_{k-1} \times F_{k-1}$ **do**
  if c_1 *and* c_2 *satisfy Equation 1* **then**
    for $i \in \{1, \ldots, f\}$ **do**
      $SIG(h_i(c_1)) \leftarrow$ first m indices in $H_i(c_1)$
      $SIG(h_i(c_2)) \leftarrow$ first m indices in $H_i(c_2)$
      $SIG(h_i(c_1 \cup c_2)) \leftarrow m$ smallest elements in $SIG(h_i(c_1)) \cup SIG(h_i(c_2))$;
      Calculate $\widehat{J_i(c_1, c_2)}$ based on Equation 5
    $\widehat{J_{\text{hybrid}}(c_1, c_2)} \leftarrow \sum_{i=1}^{f} \frac{\widehat{J_i(c_1, c_2)}}{f}$
    if $\widehat{J_{\text{hybrid}}(c_1, c_2)} \cdot (S_{c1} + S_{c2}) \geq \alpha$ **then**
      $c \leftarrow c_1 \cup c_2$
      order elements in c
      append c to C_k
return C_k

$\mathcal{D}_U$ is sparse and contains large number of items and transactions, which exactly suits our practical usage.

The *Apriori-gen* function in Algorithm 5 uses $F_{k-1} \times F_{k-1}$ method [27] to generate, C_k, the set of candidates for frequent k-item-sets. *Apriori-gen* function then uses a new, minHash-based [9] pruning technique to drastically reduce the candidates in C_k and to bring C_k as close to F_k as possible. Minimizing the cost of reducing C_k to F_k is key in achieving much higher performance than previous Apriori-based techniques.

$F_{k-1} \times F_{k-1}$ method was first systematically described in [27]. The method basically merges a pair of frequent $(k-1)$-item-sets, F_{k-1}, only if their first $k-2$ items are identical. Suppose $c_1 = \{m_1, \ldots, m_{k-1}\}$ and $c_2 = \{n_1, \ldots, n_{k-1}\}$ be a pair in F_{k-1}. c_1 and c_2 are merged if:

$$m_i = n_i \text{ (for } i = 1, \ldots, k-2), \quad \text{and} \quad m_{k-1} \neq n_{k-1}. \tag{1}$$

The $F_{k-1} \times F_{k-1}$ method generates $O\left(|F_{k-1}|^2\right)$ number of candidates in C_k. The merging operation does not guarantee that the merged k-item-sets in C_k are all frequent. Determining F_k from the usually much larger C_k becomes a major cost in Apriori execution.

Can one efficiently determine if $c \in F_k$ for any $c \in C_k$? This is the question people have been trying to directly address. However, based on the $F_{k-1} \times F_{k-1}$ method, one can alternatively ask: *Can one efficiently determine if* $c \in F_k$ *for any* c *such that* $c = c_1 \cup c_2$ *and* $c_1, c_2 \in F_{k-1}$? Dealing with c directly basically throws away the known information about c_1 and c_2. The important question then becomes how can c_1 and c_2 help determine the candidacy of c.

The key clue lies in $S(c)$, the support set of c. $S(c) = S(c_1) \cap S(c_2)$. From previous research, pruning based on the cardinality of $S(c)$ is very expensive. Instead, we propose to consider the Jaccard similarity coefficient [28] in the *Apriori-gen* function:

$$J(c_1, c_2) = \frac{|S(c_1) \cap S(c_2)|}{|S(c_1) \cup S(c_2)|}. \tag{2}$$

Measuring $J(c_1, c_2)$ is just as costly, so *Apriori-gen* uses minHash algorithm to propose a novel estimator for $J(c_1, c_2)$.

minHash scheme is a way to estimate $J(c_1, c_2)$ without counting all the elements. The basic idea in minHash is to apply a hash function h, which maps IDs to integers, to the elements in c_1 and c_2. Then $h_{\min}(c_{1/2})$ denotes the minimal hash value among $h(i), \forall i \in c_{1/2}$. Then we claim:

$$\Pr\left(h_{\min}(c_1) = h_{\min}(c_2)\right) = J(c_1, c_2). \tag{3}$$

The above claim is easy to confirm because $h_{\min}(c_1) = h_{\min}(c_2)$ happens if and only if $h_{\min}(c_1 \cap c_2) = h_{\min}(c_1 \cup c_2)$. The indicator function, $\mathbb{1}_{\{h_{\min}(c_1)=h_{\min}(c_2)\}}$, is indeed an unbiased estimator of $J(c_1, c_2)$. However, one hash function is not nearly enough for constructing a useful estimator for $J(c_1, c_2)$ with reasonable variance. The original plan is to choose k independent hash functions, $h_1, \ldots, h_k$, and construct an indicator random variable, $\mathbb{1}_{\{h_{i,\min}(c_1)=h_{i,\min}(c_2)\}}$, for each. Then we can define the unbiased estimator of $J(c_1, c_2)$ as

$$\widehat{J(c_1, c_2)} = \sum_{i=1}^{k} \frac{\mathbb{1}_{\{h_{i,\min}(c_1)=h_{i,\min}(c_2)\}}}{k}. \tag{4}$$

Before the above estimator can be implemented, it is critical to realize its computational overhead in practice. Often $k = 50$ or more is chosen and the k hash functions need to be applied to each ID in the support of each candidate. At this stage, typical applications of minHash often use the single-hash variant to reduce computation. Given a hash function h

and a fixed integer k, the *signature* of c, $SIG(h(c))$, is defined as the subset of k elements of c that have the smallest values after hashing by h, provided that $|c| \geq k$. Then the unbiased, single-hash variant of Eq. 4 is

$$\widehat{J_{s.h.}(c_1, c_2)} = \frac{|SIG(h(c_1 \cup c_2)) \cap SIG(h(c_1)) \cap SIG(h(c_2))|}{|SIG(h(c_1 \cup c_2))|}, \tag{5}$$

where $SIG(h(c_1 \cup c_2))$ is the smallest k indices in $SIG(h(c_1)) \cup SIG(h(c_2))$ and can be resolved in $O(k)$.

In general, the single-hash variant is the best minHash can offer in terms of minimizing computational cost. However, one still needs to hash all elements in c_1 and c_2 before he/she can find the signatures, which would make Eq. 5 basically as costly as Eq. 2. The key step that makes minHash estimation particularly efficient in our case is to link it with the Bloom filters assumed in our framework. Testing a member u in a Bloom filter essentially requires finding several independent hash values that map u to different indices in a bit array. Since the Bloom filter indices are comparable integers, the idea here is to avoid extra hashing in minHash calculation by re-utilizing these integer hash indices. Since all user IDs in the support sets of all frequent item-sets will be tested by the same Bloom hash functions, it guarantees the availability of these hash indices.

Suppose the Bloom filter test sets f number of bits (i.e., it runs the ID through $h_1, \ldots, h_f$ for each ID, whose membership is to be tested). The direct attempt of utilizing the Bloom filter indices in minHash is simply by replacing k in Eq. 4 with f:

$$\widehat{J(c_1, c_2)} = \sum_{i=1}^{f} \frac{\mathbb{1}_{\{h_{i,\min}(c_1) = h_{i,\min}(c_2)\}}}{f}. \tag{6}$$

A potential problem with this scheme is that, to achieve reasonable accuracy in Bloom filter and minHash, the expectations on f and k are very different. Indeed, we find $f = 7$ is sufficiently good for the Bloom filter while k is usually over 20 in order for minHash to give reliable estimates.

To overcome the empirical difference between f and k, we design a f-hash hybrid approach that uses the f already calculated Bloom hash indices. Choose k to be a fixed integer such that $k > f, k = f \cdot m$, and m is also an integer. Let h_i, for $i = 1, \ldots, f$, denote the ith Bloom hash function. Then the ith signature of c, $SIG(h_i(c))$ is the subset of m elements of c that have the smallest values after hashing by h_i, provided that $|c| \geq m$. Applying the signatures to Eq. 5, we obtain f independent estimators, $\widehat{J_1(c_1, c_2)}, \ldots, \widehat{J_f(c_1, c_2)}$. Finally, the *hybrid* estimator $\widehat{J_{\text{hybrid}}(c_1, c_2)}$ is derived as

$$\widehat{J_{\text{hybrid}}(c_1, c_2)} = \sum_{i=1}^{f} \frac{\widehat{J_i(c_1, c_2)}}{f}. \tag{7}$$

In fact, Eq. 6 is a special case of the hybrid estimator. When $k = f$ and $m = 1$, Eq. 7 becomes equivalent to Eq. 6.

Further, we have

$$\begin{aligned} J(c_1, c_2) \cdot (|S(c_1)| + |S(c_2)|) &= \frac{|S(c_1) \cap S(c_2)| \cdot (|S(c_1)| + |S(c_2)|)}{|S(c_1) \cup S(c_2)|} \\ &\geq |S(c_1) \cap S(c_2)|. \end{aligned} \tag{8}$$

Since $|S(c_1) \cap S(c_2)| = |S(c)|$, it follows that $J(c_1, c_2) \cdot (|S(c_1)| + |S(c_2)|) \geq \alpha$, if $|S(c)| \geq \alpha$, where α is the min support. Replacing $J(c_1, c_2)$ with $\widehat{J(c_1, c_2)}$ gives us the

rule *Apriori-gen* uses to reduce C_k closer to F_k. Observe that *Apriori-gen* applies the rule in reverse logical order, which introduces false positives. This is why *Apriori-gen* can only reduce C_k to some superset of F_k, but not exactly F_k.

4.3.3 Mining n-item-set with SILVERBACK+

The general association mining algorithm with the proposed pruning technique is presented in Algorithm 6. Schematically, it is similar to the original Apriori, but SILVERBACK+ effectively addresses the two issues brought up earlier in this section.

The iterations of transaction scans are minimized. The columnar database enables the algorithm to only load the necessary x column at each iteration. Further, by sorting the item-sets in each candidate set C_k and sorting the items in each item-sets, we can make sure each column is loaded only once from the disk and will stay in memory for iterations of all item-set candidates, to which this column belongs.

Probabilistic candidate pruning is key in our proposed algorithm. Indeed, we already show how it can prune off the unworthy candidates. But we are equally interested in its impact to the complexity of the algorithm. In Algorithm 6, the only temporal performance impact is line 24, where the hash indices—which come for free when testing memberships with Bloom filter—are inserted in $H_1(c), \ldots, H_f(c)$, each of which is a priority queue of length $\leq m$. The temporal cost for each ID in the test of each candidate without insertions to priority

Algorithm 6: SILVERBACK+—columnar probabilistic algorithm for finding general frequent item-sets.

Input: α, minimal support, W, set of all walls, $\mathcal{D}_U$, the database of transactions
Output: O, set of all frequent item-sets

```
1  O ← {}
2  F_1 ← {x | x ∈ W, and support_x ≥ α}
3  F_2 ← Algorithm1(α, W, D_U)
4  O ← O ∪ F_1 ∪ F_2; k ← 2
5  for each c ∈ F_2 do
6  |  S_c ← support counts from Algorithm1's byproduct
7  |  H_1(c), ..., H_f(c) ← obtained from Algorithm1
8  while F_k ≠ ∅ do
9  |  k += 1
10 |  C_k ← Apriori-gen(F_{k-1}, α,
11 |                    {H_1(c), ..., H_f(c), support_c,
12 |                                  for ∀c ∈ F_{k-1}})
13 |  order elements in C_k
14 |  for each c ∈ C_k do
15 |  |  H_1(c), ..., H_f(c) ← empty ascending priority queues each with capped capacity m
16 |  |  support_c ← 0; bf ← vector of 1s
17 |  |  y ←first item in c; U_y ← IDs from y column
18 |  |  for each x ∈ c\y do
19 |  |  |  bf ← AND-mask(bf, x column Bloom filter)
20 |  |  for each u ∈ U_y do
21 |  |  |  h_1, ..., h_f ← u's indices in bf, respectively
22 |  |  |  if h_1, ..., h_f all set in bf then
23 |  |  |  |  support_c += 1
24 |  |  |  |  append h_1, ..., h_f to H_1(c), ..., H_f(c), respectively
25 |  |  if support_c ≥ α then
26 |  |  |  append c to F_k; append c to O
27 return O
```

queues would be $O(f)$. The insertions introduce an additional complexity $O(f \log m)$. In the *Apriori-gen* function, for each candidate, lines 5 and 6 cost is $O(fm)$ and line 7 cost $O(fm \log m)$ due to sorting. To claim that the temporal cost (and the spatial cost, which is bounded by temporal) is basically constant, we need to show that both f and m are small integers and the cost does not increase as the transactions or unique items increase.

The number of Bloom hash functions f is said to be 7 in previous section and it only grows logarithmically with respect to the total transactions. So $f = 10$ would be sufficient for some 1 trillion transactions. m, on the other hand, is determined by f and the minHash error rate. minHash introduces error $\epsilon \sim O(\frac{1}{\sqrt{m \cdot f}})$ to its Jaccard estimation $\hat{J}$, which is between 0 and 1. Suppose that $\epsilon < 0.06$ is satisfactory and $f = 7$, then $m = 40$ is sufficient. Further, if f increases to 10, $m = 28$ would be sufficient for achieving the same ϵ.

SILVERBACK+ is scalable and can be deployed on a cluster. The column files and Bloom filter files are distributed across the slave servers of the cluster. An index file is stored on the master server to keep track of the slave, on which a particular column file or Bloom filter is stored. A nice property of SILVERBACK+ is that only the user IDs from one column are necessary to be loaded in memory at any given moment of the execution of SILVERBACK+. This implies that the uncompressed, large column files are *never* moved from slave to slave over the network. Only the compressed strings of Bloom filters are loaded from other slaves when necessary. This property minimizes general intra-cluster I/O traffic and makes our algorithm scalable.

5 Experimental results

We now present the experiments that evaluate the proposed methodologies.

5.1 Dataset

Our data is collected from two widely used social media platforms: Facebook and Twitter. Both Facebook and Twitter are sites for individuals, groups or businesses to post content such as messages, promotions or campaigns. The user comments/tweets and user information from specific *interests* are publicly available and collected using Facebook API and Twitter API. In the experiments, the data collected over 2012 is used. Table 2 shows the size of the databases we are maintaining using the proposed infrastructure and the amount of data used in the experiments.

5.2 Errors from sampling and Bloom filter

As discussed earlier, a Bloom filter allows for false positives. In this section we discuss how different capacity sizes and false positive probabilities affect the target-driven rule calculation.

Table 2 Datasets summary statistics

Statistic	Facebook	Twitter
Unique items/interests (used in experiments)	32 K + 22,576	11 K + 4291
Total user activities (used in experiments)	10 B + 226 M	900 M + 24.2 M
Unique users/transactions (used in experiments)	740 M + 27.4 M	120 M + 3.7 M

Table 3 Samples of detailed dataset statistics

Interest	TM	CM	C1	C2	C3	C4	C5	C6	C7
EASPORTS	242,399	1647	33,197	1647	10,085	6611	1708	2136	1714
Techcrunch	202,812	12,295	32,579	12,295	17,105	15,647	12,950	13,147	12,496
iTunesMusic	189,568	7265	24,171	7265	10,625	9698	7513	7640	7640
Google	149,877	12,022	21,352	12,022	13,797	13,621	12,605	12,636	12,636
Facebook	120,724	8904	14,212	8904	9746	9859	9356	9365	9365

With the introduction of the probabilistic data structure, the computation of Supp$\{X \cup Y\}$ (i.e., the common users that have shown interests in both interests X and Y) is affected which, in turn, affects the order the relevant precise interests.

Table 3 shows the precise interests generated for target interest *amazon* for the period of July–December of 2012. For each interest we provide Total Mentions (*TM*), which is the number of users who expressed interest, Common Mentions (*CM*), which is actual number of common users who expressed interest for both interests (true positives), and different configurations of Bloom filters. Configurations $C1$, $C2$ and $C3$ have false probability 0.10, 0.002 and 0.02, respectively, and a filter capacity of 100,000. Configurations $C4$, $C5$ and $C6$ have false probability 0.10, 0.002 and 0.02, respectively, and a filter capacity of 200,000. Configuration $C7$ is the only configuration where the Bloom filter is built using sample (S) size equal to the capacity size (200,000) if the TM is over the capacity size and its false probability is 0.02. In configuration $C7$, the common mentions for the Bloom filter is then estimated proportionately based on the total mentions. Note the that total number of mentions for *amazon* is 184,117.

Due to the probabilistic nature of the data structure, we use predictive analysis approach where we evaluate the effective measure of our system by formulating a confusion matrix, i.e., a table with two rows and two columns that reports the number of false positives, false negatives, true positives and true negatives. The common mentions given by Bloom filter comprise of true positives and false negatives. Table 4 provides the number of false positive (*fp*), which deduced using common mentions from Bloom filter and true common mentions. The number of false negatives is always zero due to the nature of Bloom filter. Therefore, the true negatives (not shown in table) are easily deduced. The accuracy, precision and F-measure are also provided in Table 4.

As expected, for a given capacity, as the false positive probability decreases, the accuracy $((tp + tn)/(tp + tn + fp + fn))$ and precision $(tp/(tp + fp))$ both increase. The recall $(tp/(tp + fn))$ is always 1.0, i.e., all relevant users were retrieved because our system with Bloom filter does not permit false negatives. The precision for our system is always less than 1.0 as not every result retrieved by the Bloom filter is relevant. As the capacity is increased, the accuracy and precision further improve. Note that when the total mentions is greater than the capacity, the Bloom filter has higher inaccuracy for a fixed false probability. For example for *EASPORTS* , the accuracy is 15 % lower for capacity of size 100 versus 200 K for the false probability of 0.10. This is due to the property that adding elements to the Bloom filter never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1. To counter this effect, we sample the data to be added to Bloom filter. Sampling can have an impact on the false positive rate of Bloom filters depending on the sampling quality. For example, the number of false positives for *EASPORTS*, for Bloom filter configurations *C5* and *C7*, is 61 and 67, respectively. But the false positives drop for *techcrunch* when sampling is used.

Table 4 Bloom filter accuracy results

Interest	C1	C2	C3	C4	C5	C6	C7
False positives							
EASPORTS	31,550	1402	8438	4964	61	489	67
Techcrunch	20,284	1085	4810	3352	655	852	201
iTunesMusic	16,906	568	3360	2433	248	375	375
Google	9330	648	1775	1599	583	614	614
Facebook	5308	469	842	955	452	461	461
Accuracy							
EASPORTS	0.829	0.992	0.954	0.973	1.000	0.997	1.000
Techcrunch	0.890	0.994	0.974	0.982	0.996	0.995	0.999
iTunesMusic	0.908	0.997	0.982	0.987	0.999	0.998	0.998
Google	0.949	0.996	0.990	0.991	0.997	0.997	0.997
Facebook	0.971	0.997	0.995	0.995	0.998	0.997	0.997
Precision							
EASPORTS	0.050	0.540	0.163	0.249	0.964	0.771	0.961
Techcrunch	0.377	0.919	0.719	0.786	0.949	0.935	0.984
iTunesMusic	0.301	0.927	0.684	0.749	0.967	0.951	0.951
Google	0.563	0.949	0.871	0.883	0.954	0.951	0.951
Facebook	0.627	0.950	0.914	0.903	0.952	0.951	0.951
F-measure							
EASPORTS	0.095	0.701	0.281	0.401	0.982	0.569	0.980
Techcrunch	0.548	0.958	0.836	0.893	0.974	0.932	0.992
iTunesMusic	0.462	0.962	0.812	0.881	0.983	0.929	0.975
Google	0.720	0.974	0.931	0.952	0.976	0.964	0.975
Facebook	0.770	0.974	0.955	0.964	0.975	0.970	0.975

Table 5 Kendall τ rank correlation table

Measure	200 K, 0.02	200 K, 0.002
Kendall τ-statistic	0.98251	0.98455
Two-sided p value	$<$0.00001	$<$0.00001
S, Kendall score	3847	3855
Var (S)	79,624.33	79,624.34
S/τ, denominator	3915.5	3915.5

Due to probability of false positives, the interests order arranged in decreasing order of the common mentions count can be different. We use the Kendall rank correlation coefficient or short for Kendall's tau (τ) coefficient [15] to evaluate our results. Measuring the rank difference instead of absolute error that our probabilistic algorithm is due to practical interests in a rank-oriented output. It is more often the case that our customers would ask queries such as the *top X number of frequent items* associated with my brand. A similar rank-oriented objective is proposed and receives good feedback in the literatures of recommender systems (detailed in section 4.1 of [20]). τ is defined as the ratio of the difference between

Table 6 Effect of using hierarchical algorithm to speed up the mining of frequent 2-item-sets

Setup	Alternative	Baseline	Speedup
(100, 10, 100 K, .1)	27.6	142	5.14
(100, 10, 100 K, .01)	54.1	136	2.52
(1 K, 10, 1 M, .01)	599	923	1.541
(1 K, 50, 10 M, .01)	1371	3038	2.216
(10 K, 50, 100 M, 10^{-4})	753	4169	5.536
(10 K, 100, 200 M, 10^{-5})	1756	6955	3.691
(100 K, 1 K, 1 B, 10^{-6})	778	38,080	48.95
(200 K, 2 K, 2 B, 10^{-6})	2654	193,035	72.74

Table 7 Parameters for results shown in Fig. 4

Linking metric	Unique items	Total transactions	User sparsity	Pruning strategy
Random or 2D support	128 or 256	20 M or 40 M	10^{-3} or 10^{-6}	Aggressive or conservative

concordant and discordant pairs to the total number of pair combinations. The coefficient range is $-1 \leq \tau \leq 1$, where 1 implies perfect agreement between rankings. Table 5 provides the Kendall statistics for two Bloom filter configurations. Both configurations approximately have τ value of 0.98, implying that our rankings are very close in agreement compared to original rank. Also since the two-sided p value is less than 0.00001, this implies that the two orderings are related and the τ values are obtained with almost 100 % certainty. Further discussions about Kendall statistics are presented in our previous work [32].

5.3 Comparing two 2-item-set mining algorithms

In Table 6, we show the speedup of 2-item-set mining by using the hierarchical algorithm (see Algorithm 2) from Apriori-based algorithm (see Algorithm 1). The baseline time is obtained by running Apriori-based algorithm and alternative time is obtained by using the hierarchical algorithm. We have done so by reporting several different setups, which are presented in the first column of Table 6 as a formatted tuple: (number of items in W, largest column size in W, total transactions, average column similarity in W). Table 7 and Fig. 4 give more insights into how the benefits of the hierarchical algorithm with respect to different parameters and data sizes. Table 7 lists out the parameters and the different values that each parameter can take. "Link metric" means the metric M used in the linkage() function from Algorithm 2. "2D support" refers to $M = \{(\text{support}(i), \text{support}(i)^2) | i \in W\}$ and "Random" simply means $M = \{\text{random number} | i \in W\}$. "Unique items" and "Total transactions" describe the size of transaction datasets. "User sparsity" describes how likely for the items to appear in a same user's transaction. A lower sparsity level usually indicates smaller transaction size. Finally, "Pruning strategy" refers to the test condition in Algorithm 4. "Conservative" implements the condition shown in line 7 of Algorithm 4 and "Aggressive" implements the heuristic strategy mentioned earlier: "**if** $count \geq \alpha$ **and** $count \geq |F|$".

Figure 4 shows various performance figures. The triple tuples on x-axis encode the ("Unique items","Total transactions" in millions, "User sparsity" in $\log_{10}$) configuration. Five series are shown in Fig. 4: "linear base" is the baseline Apriori-based algorithm with

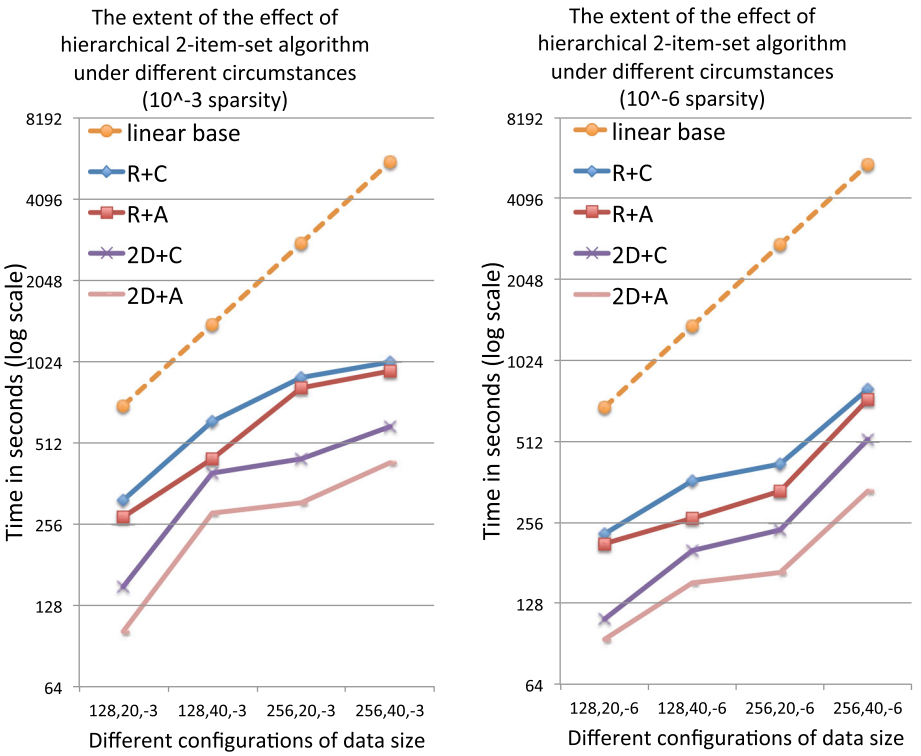


Fig. 4 Effect of using hierarchical binary tree of Bloom filters. For ease of reading, the order of the legends is sorted to match the stacking order of the lines (best viewed in *color*) (color figure online)

ideal projected scalability; "R+C" is the hierarchical algorithm using random link metric and conservative pruning strategy; "2D+A" is the hierarchical algorithm using 2D support link metric and aggressive pruning strategy; etc. Results from Fig. 4 suggest that using "2D support" as the linkage metric and the "Aggressive" pruning strategy can further improve the performance of our proposed hierarchical 2-item-set mining algorithm.

5.4 Scalability in n-item-set mining ($n > 2$) algorithms

In addition to evaluating the accuracy of our probabilistic algorithms, we still need to demonstrate their efficiency and scalability. After all, good efficiency and scalability are expected trade-offs by sacrificing accuracy.

In Fig. 5, we report the runtimes for different combinations of computing nodes, and minimum support threshold values, for four different algorithms. In the legend of Fig. 5, HA denotes the naive implementation of Apriori in the MapReduce framework [22]. CS, CSBF and SILVERBACK+ denote our proposed algorithm with progressively more features. CS denotes a diminished version, where only the columnar storage is used but not the Bloom filter enhancement or the minHash pruning technique; CSBF is like CS but implements the Bloom filter enhancement for each column file; and finally, SILVERBACK+ is the fully blown version that incorporates all techniques presented in our paper including the minHash pruning technique. In addition, a dashed line of ideal scalability is included for each of the four methods compared in Fig. 5.

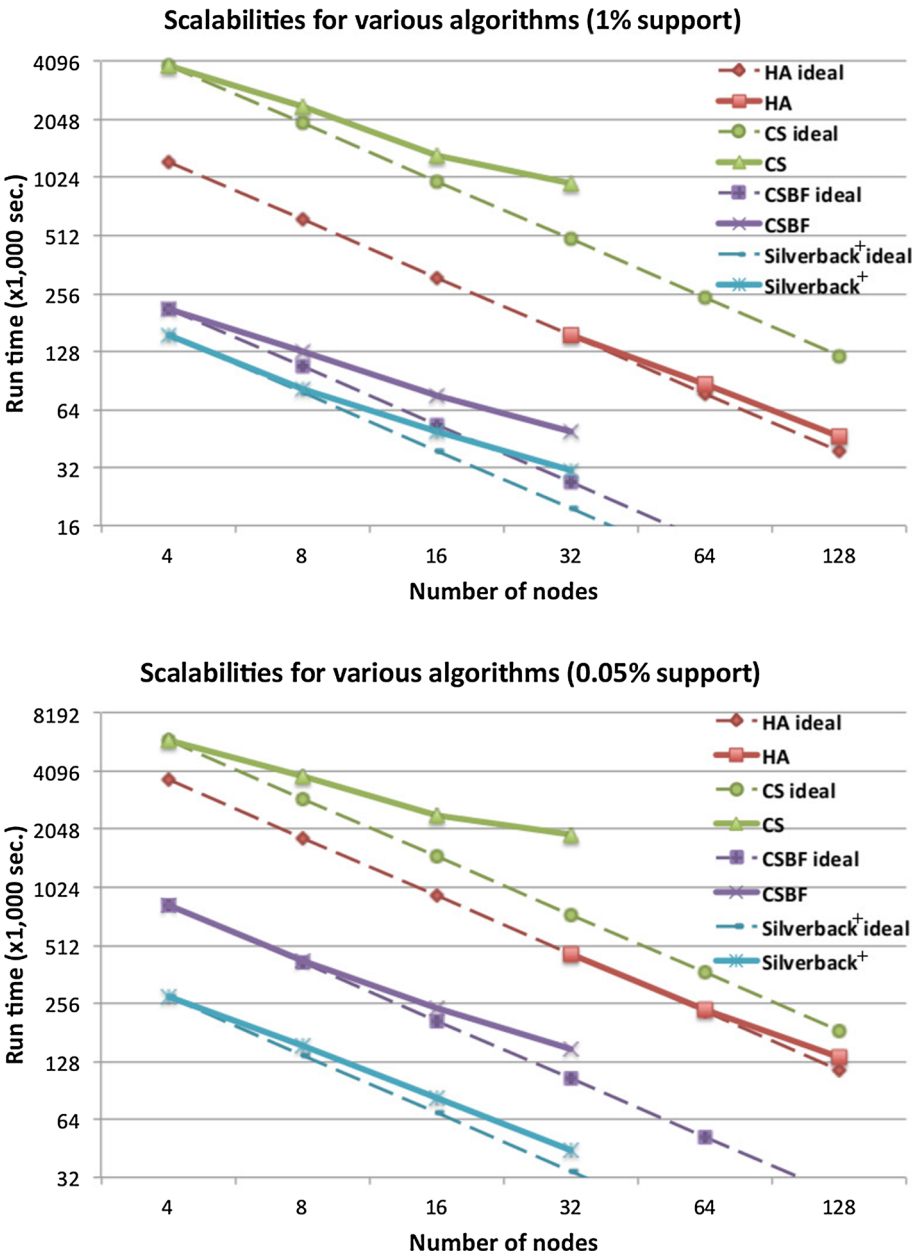


Fig. 5 Scalability comparison of four methods (HA, CS, CSBF and SILVERBACK+). Each method is *color coded* separately (best viewed in *color*) and is accompanied by a *dashed line* of the *same color* which extrapolates the runtime if the runtime were to decrease linearly with respect to the number of nodes (color figure online)

In both support levels (0.05 and 1 %), HA seems to have the most reliable speedup as the number of computation nodes increases. The CS method significantly deviates from the ideal speedup as we increase up to 32 nodes. We suspect its lack of scalability is due to the increase

of I/O traffic, since the IDs in each column are not compressed like CSBF or SILVERBACK+ and would pose significant load on the I/O. Both CSBF and SILVERBACK+ exhibit superior scalability over CS, especially in the low support setup.

HA, the Hadoop solution, seems to have better scalability than all other algorithms, although its absolute runtime is not the lowest. Will HA be the fastest eventually if the number of nodes keeps on increasing? We think the relatively superior scalability in HA is mainly due to two aspects. First, HA, unlike the other three methods, is implemented on a Hadoop cluster with slightly better computational capability per node but *much* better inter-node connections (32 Gbit/s InfiniBand). The budget cluster, on which CS, CSBF and SILVERBACK+ are implemented, simply uses corporation-domain IP addresses as node identifiers. Second, SILVERBACK+ still has room to improve its scalability to more nodes as this algorithm is only proposed in this paper while Hadoop Apriori is much more mature.

The ranks of performance for the four methods are consistent under both support levels. The two probabilistic approaches, CSBF and SILVERBACK+, perform consistently faster than the exact ones, HA and CS, which is predicted as we expect sacrificing accuracy would significantly boost the temporal performance. CS performs consistently worst, which suggests that proposing a columnar storage by itself does not quite solve any problem.

Investigating the relative changes in the inter-method gaps under different support levels reveals more on the impact of minHash pruning and Bloom filter enhancement. First, the difference made by using Bloom filters, as illustrated by CS and CSBF, increases when min support level drops. Second, the use of minHash pruning technique also amplifies its impact as the support level decreases.

6 Conclusions

We presented the SILVERBACK+ framework—an end-to-end solution for association mining from large-scale social behavioral databases under constraints of modest hardware. We proposed accurate probabilistic algorithms for mining frequent item-sets, specifically catering to the columnar storage that we adopted, which is enhanced by Bloom filters and reservoir sampling techniques to enable storage efficiency. For basic frequent 2-item-set mining, we proposed a novel algorithm based on a hierarchical binary tree of Bloom filters, which performs considerably better than Apriori-based solutions. For frequent n-item-set ($n > 2$) mining, we introduced an Apriori-based algorithm that prunes candidate item-sets without counting every candidate's support. As our experiments showed, SILVERBACK+ outperforms Hadoop Apriori on a more powerful cluster in terms of a runtime, with a probabilistic approach yielding a satisfactory accuracy. Overall, we show that leveraging multiple existing solutions, combing them judiciously, and adding innovative core details can significantly outperform the "go-to" option.

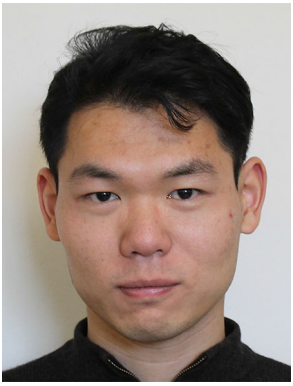
The SILVERBACK+ framework has been successfully deployed and maintained at 4C, a social media mining startup, since May 2011. Our ongoing efforts are focusing on further improvements of our system performance and scalability—specifically, we are developing more efficient inter-nodal communication solutions, which is critical to scale to hundreds of nodes. From a broader perspective, we are investigating the applicability of our ideas of approximate association mining in different contexts [11,17].

Acknowledgements This work is supported in part by the following Grants: NSF awards CCF-1029166, IIS-1343639, CCF-1409601, CNS-0910952 and III 1213038; DOE awards DE-SC0007456, DE-SC0014330; ONR Grant N00014-14-1-0215.

References

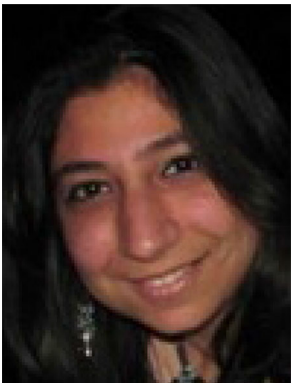
1. Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. In: SIGMOD'93. ACM, pp 207–216
2. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules in large databases. In: Proceedings of the VLDB Endow, VLDB'94, pp 487–499
3. Bayardo RJ Jr (1998) Efficiently mining long patterns from databases. In: SIGMOD'98. ACM, New York, NY, USA, pp 85–93
4. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors, vol 13. ACM, New York, pp 422–426
5. Cao H, Wolfson O, Trajcevski G (2006) Spatio-temporal data reduction with deterministic error bounds. VLDB J 15(3):211–228
6. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2006) Bigtable: a distributed storage system for structured data. In: OSDI'06. USENIX Association, pp 15–15
7. Chen J, Stallaer J (2014) An economic analysis of online advertising using behavioral targeting. MIS Quarterly 38(2):429–449
8. Chung S, Luo C (2003) Parallel mining of maximal frequent itemsets from databases. In: ICTAI'03, pp 134–139
9. Cohen E, Datar M, Fujiwara S, Gionis A, Indyk P, Motwani R, Ullman JD, Yang C (2001) Finding interesting associations without support pruning, vol 13. IEEE, pp 64–78
10. Cormode G, Garofalakis MN (2008) Approximate continuous querying over distributed streams. ACM Trans Database Syst 33(2):1–39
11. Grupcev V, Yuan Y, Tu Y-C, Huang J, Chen S, Pandit S, Weng M (2013) Approximate algorithms for computing spatial distance histograms with accuracy guarantees. IEEE Trans Knowl Data Eng 25(9):1982–1996
12. Han J, Pei J, Yin Y (2000) Mining frequent patterns without candidate generation. In: SIGMOD'00. ACM, pp 1–12
13. Hofmann T, Buhmann J (1997) Pairwise data clustering by deterministic annealing, vol 19. IEEE, pp 1–14
14. Kallman R, Kimura H, Natkins J, Pavlo A, Rasin A, Zdonik S, Jones EPC, Madden S, Stonebraker M, Zhang Y, Hugg J, Abadi DJ (2008) H-store: a high-performance, distributed main memory transaction processing system, vol 1, VLDB Endowment, pp 1496–1499
15. Kendall M (1938) A new measure of rank correlation, vol 30. Biometrika Trust, pp 81–93
16. Kimura N, Latifi S (2005) A survey on data compression in wireless sensor networks. In: ITCC (2), pp 8–13
17. Kumar A, Grupcev V, Yuan Y, Huang J, Tu YC, Shen G (2014) Computing spatial distance histograms for large scientific data sets on-the-fly, vol 26. IEEE, pp 2410–2424
18. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system, vol 44. ACM, New York, pp 35–40
19. Lan B, Ooi BC, Tan K-L (2002) Efficient indexing structures for mining frequent patterns. In: ICDE'02, pp 453–462
20. Lee J, Bengio S, Kim S, Lebanon G, Singer Y (2014) Local collaborative ranking. In: Proceedings of the 23rd international conference on World Wide Web. In: WWW'14. ACM, New York, NY, USA, pp 85–96
21. Li H, Wang Y, Zhang D, Zhang M, Chang E (2008) Pfp: parallel fp-growth for query recommendation. In: RecSys'08, pp 107–114
22. Lin M-Y, Lee P-Y, Hsueh S-C (2012) Apriori-based frequent itemset mining algorithms on mapreduce. In: ICUIMC'12
23. Ozkural E, Aykanat C (2004) A space optimization for FP-growth. In: FIMI
24. Pu IM (2006) Fundamental data compression. Elsevier, Amsterdam
25. Qiu L, Li Y, Wu X (2007) Preserving privacy in association rule mining with Bloom filters. J Intell Inf Syst 29(3):253–278
26. Sparse matrices (2014) http://docs.scipy.org/doc/scipy/reference/sparse.html
27. Tan P-N, Steinbach M, Kumar V (2005) Introduction to data mining, 1st edn. Addison Wesley, Reading
28. Turrisi R, Jaccard J (2003) Interaction effects in multiple regression, vol 72. Sage, London
29. Vitter JS (1985) Random sampling with a reservoir, vol 11. ACM, New York, pp 37–57
30. Xie Y, Chen Z, Zhang K, Patwary M, Cheng Y, Liu H, Agrawal A, Choudhary A (2013) Graphical modeling of macro behavioral targeting in social networks. In: SDM, pp 740–748
31. Xie Y, Cheng Y, Honbo D, Zhang K, Agrawal A, Choudhary AN, Gao Y, Gou J (2012) Probabilistic macro behavioral targeting. In: DUBMMSM, pp 7–10

32. Xie Y, Palsetia D, Trajcevski G, Agrawal A, Choudhary AN (2014) Silverback: scalable association mining for temporal data in columnar probabilistic databases. In: ICDE, pp 1072–1083
33. Ye Y, Chiang C-C (2006) A parallel apriori algorithm for frequent itemsets mining. In: SERA'06. IEEE, pp 87–94
34. Zaki MJ (2000) Scalable algorithms for association mining, vol 12. IEEE Educational Activities Department, Piscataway, pp 372–390
35. Zaki MJ, Parthasarathy S, Li W (1997) A localized algorithm for parallel association mining. In: SPAA'97, pp 321–330



Yusheng Xie received B.S. (Summa Cum Laude) in Computer Engineering and Applied Mathematics from Northwestern University, Evanston, IL, in 2011. He obtained his PhD degree in Computer Engineering at Northwestern University in 2015. He is now with Baidu Research in Sunnyvale, California.

Zhengzhang Chen received his PhD in Computer Science from North Carolina State University, Raleigh, NC, in 2012. Dr. Chen was a Research Assistant Professor at Northwestern University from 2012 to 2014. He is now with NEC Laboratories America.



Diana Palsetia is a data scientist at 4C in Chicago, Illinois. She received her bachelor's degree in Computer Engineering from Michigan Technological University, her master's degree in Electrical Engineering from University of Wisconsin–Madison, and her PhD from Northwestern University.

Goce Trajcevski is currently an Assistant Chair with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL. His research has been funded by BEA, Northrop Grumman Corp., National Science Foundation and Office of Naval Research. He received Best Paper Awards at CoopIS 2000 and MDM 2010, and Best Short-paper Award in MSWiM 2013.

Ankit Agrawal is a Research Associate Professor in the Department of Electrical Engineering and Computer Science at Northwestern University. He got his PhD from Iowa State University in 2009 and B.Tech from Indian Institute of Technology (IIT) Roorkee in 2006. He is serving as a PI/Co-PI on funded research grants from NSF, DOE, AFOSR, NIST and DARPA.

Alok Choudhary received PhD in electrical and Computer Engineering from the University of Illinois, Urbana-Champaign, in 1989. Since 2000, he has been a Professor at Northwestern University, Evanston, IL. Prof. Choudhary is a fellow of IEEE, ACM and AAAS. Follow @alokchoudhary01 on Twitter.