SILVERBACK: Scalable Association Mining For Temporal Data in Columnar Probabilistic Databases

Yusheng Xie [#1], Diana Palsetia [*2], Goce Trajcevski [*3], Ankit Agrawal [*4], Alok Choudhary [#5]

[#] *Voxsup Inc.,*
79 W Monroe St., Chicago, IL USA
[1] yves@voxsupinc.com
[5] alok@voxsupinc.com

[*] *EECS Department, Northwestern University*
2145 Sheridan Rd., Evanston, IL USA
[2] drp925@eecs.northwestern.edu
[3] goce@eecs.northwestern.edu
[4] ankitag@eecs.northwestern.edu

***Abstract*—We[1] address the problem of large scale probabilistic association rule mining and consider the trade-offs between accuracy of the mining results and quest of scalability on modest hardware infrastructure. We demonstrate how extensions and adaptations of research findings can be integrated in an industrial application, and we present the commercially deployed SILVERBACK framework, developed at Voxsup Inc. SILVERBACK tackles the storage efficiency problem by proposing a probabilistic columnar infrastructure and using Bloom filters and reservoir sampling techniques. In addition, a probabilistic pruning technique has been introduced based on Apriori for mining frequent item-sets. The proposed target-driven technique yields a significant reduction on the size of the frequent item-set candidates. We present extensive experimental evaluations which demonstrate the benefits of a context-aware incorporation of infrastructure limitations into corresponding research techniques. The experiments indicate that, when compared to the traditional Hadoop-based approach for improving scalability by adding more hosts, SILVERBACK – which has been commercially deployed and developed at Voxsup Inc. since May 2011 – has much better run-time performance with negligible accuracy sacrifices.**

I. INTRODUCTION

Behavioral targeting refers to techniques used by advertisers whereby they can reach target audience by specific interests and/or users' activity history.

To increase the effectiveness of their campaigns, advertisers employ *behavioral targeting* of customers, by capturing data generated by user activities. In the context of social websites behavioral data [1], [2] is generated in the form of likes, posts, retweets, or comments – however its foremost characterization is the large volume. For example in March 2012, nearly 1 billion of public comments or post likes were generated by Facebook users alone, according to our estimation.

Mining valuable knowledge from behavioral data relies on the data mining techniques developed for more traditional data sources. However, it turns out 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. 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, an interesting task would be finding a list of relevant interests that share a similar demographic. As it may be observed, this operation is analogous to finding frequent item-sets and association rules from a large number of transactions of co-occurrences of the items [3].

The *scale* of the data in the online behavioral world is one factor posing challenges of a different nature, with respect to 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.

At the heart of the motivation for this project are the following two, in some sense, complementary observations:
(1) Adding more hardware could help addressing the scalability – however, what if a Big Data startup cannot afford this "brute force" avenue of attaining sufficient computing power? Contrary to large enterprizes like Facebook or Twitter, many of their smaller-in-scale partner startups have few database engineers challenged with designing a system that could handle inundating amount of data sent from their larger social network partners. Constraints on the budget and even considerations for energy-saving call for designing alternatives to the simple "put it on more machines and scale" approach, and are dictating careful designs on commodity hardware.
(2) The utility of extracted knowledge from the large scale behavioral data may be improved by proper exploitation of statistical techniques. Probabilistic approaches, for as long as they do not affect the accuracy of mining results past certain degree of quality assurance, do seem like viable avenues towards efficient storage schemes.

Our main contribution presented in this paper is SILVERBACK – a probabilistic framework for accurate association rule and frequent item-set mining at massive streaming scale,

[1] Yusheng Xie and Diana Palsetia contributed equally.

implemented on a commodity hardware. It relies on a column-based storage for managing large database of transactions, incorporating Bloom filters and appropriate sampling techniques to yield faster probabilistic database while maintaining satisfactory accuracies. Complementary to this, we develop an Apriori [4], [5] based algorithm to probabilistically prune candidates without support-counting for every candidate item-set. Our experimental findings demonstrate that SILVERBACK is significantly more efficient than a generic MapReduce implementation. 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.

In the rest of this paper, after brief preliminaries and problem formulation (Section II), in Section III we position the work with respect to the related literature and present some observation justifying our approach. Sections IV and V address in greater detail the storage and infrastructure, as well as (versions of) algorithmic designs. In Section VI we present our comprehensive experimental observations, and in Section VII we conclude the work and outline directions for future work.

II. Problem Formulation

We now give a more formal specification of the tasks addressed in this work.

Given large databases of users' activity log, the challenge in our application is to parsimoniously and accurately compute target-driven frequent item-sets and association rules, and provide a real-time on-demand response.

Let $\mathcal{D}$ denote a (large) list of users' activities across public walls in the Facebook network (or handles from Twitter), consisting of quadruples $(u_i, w_i, t_i, a_i) \in \mathcal{D}$ $(i = 0, 1, \ldots, |\mathcal{D}|)$ denoting individual user's activity. The interpretation is that for i-th 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, $|W| << |U| << |\mathcal{D}|$. Therefore, it is entirely expected that $u_i = u_j$ or $w_i = w_j$ for some $i \neq j$.

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 transaction 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.

The goal, given a live and rapidly growing $\mathcal{D}$ and a target Y, 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.

III. Related Work

This section gives an overview of a body of relevant works and casts our work in that context.

A. Association Mining

Association Mining aims at finding correlations between items in a dataset. And despite the recent advances in parallel association mining algorithms [6] [5], the core technique is largely unmodified. The popular Apriori [4] algorithm identifies the frequent items by starting with small item-sets, and only proceeding to larger item-sets if all subsets are frequent – incurring a cost-overheads because in every count step it scans the entire database. 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:

1) Max-Miner: Max-Miner [10] 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 two item-sets. First called *head* ($h(g)$), which is the item-set enumerated by the node. 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 sub-nodes 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 sub-node will also be frequent but not maximal, and therefore the sub-node expansion can be halted. If $h(g) \cup \{i\}$ is infrequent then any head of a sub-node that contains item i is infrequent. Consequently, subset pruning can be implemented by removing any such tail item from candidate group before expanding its sub-nodes.

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) Frequent Pattern (FP) Growth: FP-Growth [11] 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 data-set, and frequent item-sets are discovered by traversing through the FP-tree.

TABLE I
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	Many	Large	Good[6]	benchmark	Yes	∼1,000	Exact
Max-Miner	Row-based	Row-based	Many	Large	Fair[7]	$\sim 5\times$	Yes	Unknown	Exact
Eclat	Columnar	Flexible	A few	Small	Poor[8]	$3\times \sim 10\times$	Yes	∼2,000	Exact
FP-Growth	Row-based	FP tree	2	Enormous	Very Good[9]	$5\times \sim 10\times$	Yes	7,000+	Exact
SILVERBACK	Columnar	Flexible	A few	Tiny	Good	$> 15\times$	Const. time	∼2,000	Probabilistic

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 that share common prefix item, 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 sub-tree 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 data-set is added to FP-Tree.

3) Eclat: Similarly to FP-growth, Eclat employs the divide and conquer strategy to decompose the original search space [12]. 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.

4) Distributed and Parallel Algorithms: Discovering patterns from a large transaction data set 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 [8], Parallel Max-miner [7], Parallel FP-Growth [9] and Distributed Apriori [6].

Table I compares our proposed method with other popular existing methods in many aspects including their scalability to more nodes.

B. Modern Applications of Bloom Filters

Capturing demographic between any two interests can be very high in space complexity as it requires membership operation to be performed. Bloom filter is a popular space-efficient probabilistic data structure used to test membership of an element [13]. For example, Google's BigTable storage system uses Bloom filters to speed up queries, by avoiding disk accesses for rows or columns that don't exist [14]. 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 either to add items or to check whether an item is in the set is fixed – $O(k)$, where k is the number of hash functions – independent of the number of items already in the set. The caveat, though, is that it 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: $m = -n \ln p/(\ln 2)^2$.

C. OLAP and Data Warehouse

Traditional OLAP (On-Line Analytical Processing) queries are usually generated by aggregation along different spatial and temporal dimensions at various granularities. For example, OLAP queries for a typical job posting website [15] include *job views by day/week/month*, *job views by city*, and *job views by company*. For efficiency, OLAP queries are issued against specifically designed data warehouses. Current industrial practices usually build a data warehouse in three steps: (1) cubifying the raw data; (2) storing the cubes; and (3) mapping OLAP queries into cube-level computations.

Web-scale real-time applications like Twitter and Facebook pose tremendous challenges to the three straightforward steps of building data warehouses.

Insights-seekers basically demand real-time summary statistics about the website at any granularity. To support such stringent demands, a modern data warehouse is typically first built from existing log data and incrementally updated by setting up a, so called, *river* – a persistent link between source and destination carrying real-time data stream. In addition, powerful and elastic MapReduce frameworks like Hadoop [16] are usually deployed to handle the first step, the cubification of web-scale data, and the third step, mapping queries to

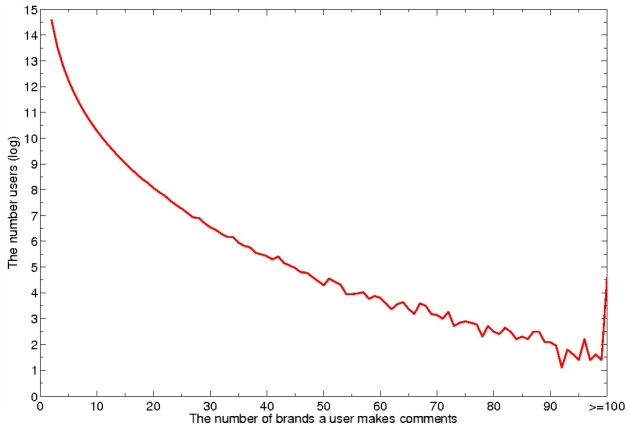

Fig. 1. Facebook user activity distribution (June 2008 to January 2012, with vertical axis in log scale)

cube-level computations. Durable and scalable key/value pair storages like Cassandra [17] are often necessary to fulfill the second step of building a web-scale data warehouse. In short, more hardware and scalability seem to be the two hosts that keep the system going – which, as indicated in Section I, is something that we aim to change.

IV. Storage and Infrastructure

Given the scale of $\mathcal{D}_U$ in our settings, the traditional row-based storage assumed by [4] [11] [18] would become out of depth. Our objective is to provide efficient storage scheme, however, we are not trying to invent a general-purpose advanced distributed storage engine to add to the already abundant list of such engines and file systems. Instead, we focus on an application/data-driven ad-hoc solution and we discover that a probabilistic column storage is very effective in tackling the massive data scale in our application domains.

A. Scalable Column Storage

The key observation that motivated our design is the sparsity of $\mathcal{D}_U$. The full representation of $\mathcal{D}_U$ would require over 20 trillion cells (740M users by 32K walls), which is impractical even in distributed environment (notwithstanding the budget). However, of the 20 trillion cells, less than 1% are populated. According to our estimates, an average user accesses less than 14 of the 32K walls. The sparsity of $\mathcal{D}_U$ is neither a coincidence nor a surprise us – in fact, the global sparseness in a social graph and the power-law decay in its node degree distribution are part of the asymptotic behavior that we can safely assume. As an illustration, Figure 1 shows the distribution of 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. We 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 only contain a small number of items.

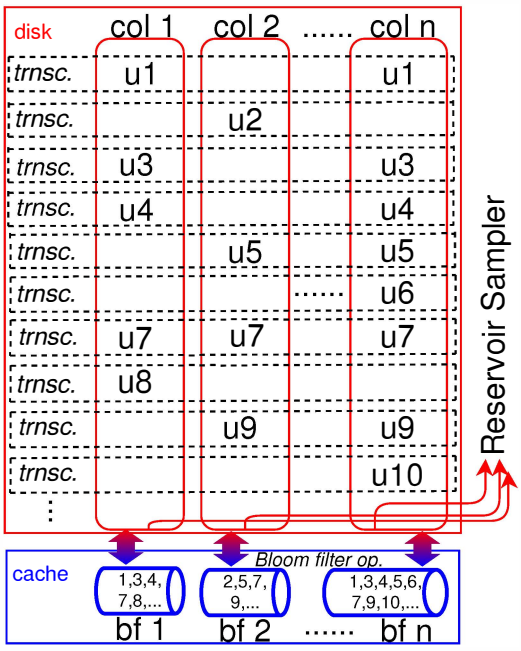

Fig. 2. Illustration of the columnar storage in place of traditional row-based transactions(trnsc.) and its probabilistic enhancement

We use a sparse representation of the massive $\mathcal{D}_U$ called "list of lists" (LIL) [19] (or "Column Family" in Cassandra [17]). LIL typically stores a massive sparse matrix by using a list to record the non-zero 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 Figure 2 illustrates how the traditional row-based transactions of items in a database are stored as columns. The LIC implementation is popular among columnar databases.

One of the advantages in this columnar storage is data independency. The LIC representation of the database $\mathcal{D}_U$ can be partitioned by columns and we can store the columns 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 particularly helpful when the database is live like $\mathcal{D}_U$.

B. 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 like 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. The situation would be much easier to deal with if all columns are similar in size, which would allow the allocator to treat

all columns equally. To tackle this problem, 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 independency, 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 [20]. But MongoDB 2.1 generically implements readers-write lock and allows one write queue per database, which is not desirable in our case and may have unforeseeable impact at large scale.
(2) Another approach – which we adopted as our philosophy is to simply solve the locking problem by "avoiding it". Namely, similar to [21], 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 [22] 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.

However, the main problem now becomes that the column files still cannot fit into the main memory of our modest cluster, even after sampling – we note that loading all column files of the described $\mathcal{D}_U$ requires roughly 300GB after sampling. The practical goal is to reduce the representation of $\mathcal{D}_U$ from 300GB down to approximately 25GB – without breaking data independency, performance or scalability. With such constraints, our options are limited due to "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 [13] with its probabilistic storage-efficiency seems a plausible choice. A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. Hence, our idea is to construct a bloom filter for each column, as depicted in the bottom part in Figure 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.

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 854.5KBytes $\times\ 30,000 < 24.5$ 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 7 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/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 we will demonstrate in Section VI, both the sampling and Bloom filter have a very limited impact on the accuracy of the results.

C. Deployment of SILVERBACK

The commercially deployed SILVERBACK system consists of three major parts: (1) columnar probabilistic database of transactions; (2) a computation cluster; and (3) storage for output rules and frequent item-sets.

The database of transactional records, $\mathcal{D}$, is implemented using modified versions of MySQL [23] and MongoDB [20] on top of 6 relatively powerful nodes. Since the database infrastructure is shared with other data warehousing purposes, databases are served from dedicated servers (free of other computational chores) to achieve high I/O throughput.

The computation nodes are the ones executing the SILVERBACK mining algorithms (cf. Section V), implemented as web services and served from scalable web servers like Tornado [24]. Therefore, most communication between the database and the computation cluster is through internal HTTP requests. About 30 nodes are deployed in this cluster, which is a shared resource among several computation-intensive purposes including association mining. The cluster is logically organized as master server, shadow master servers for fault-tolerance, and slave servers. However, physically several slave servers can reside on a same actual node; moreover, the master server is run alongside with slave servers on a same node as well. All the slave servers are designed to recover from crash and resume from its last checkpoint.

Two important design decisions in our computation infrastructure are: (1) implementing the computation as web service-based transactions; and (2) the "ideological" separation between logical servers and physical nodes.

A substantial advantage of turning computation tasks into service-based transactions is the elimination of startup cost of loading dictionaries, lookup tables from disk, since the end points for those web services are persistent. More specifically,

the Bloom filter structures, which are small in memory footprint, once fit into the main memory of the web servers can be tested, copied, and updated without modifying the disk as long as the hosting web services do not restart themselves. Service-based system also makes logging much easier and can be readily integrated with frameworks like Scribe [25]. Another advantage, of a particular interest for our commercial application, is the web-servers' built-in handling for timeout requests. Suppose the system is calculating frequent item-sets on-the-fly from end-clients' requests. Often, the desideratum is not to find complete/exact frequent item-sets in as efficiently (in time) as possible. To the contrary, the clients expect to explore as many frequent item-sets as possible after a tolerably short time-delay, say, 1 second. Service-based implementation makes it easier to achieve such expectations.

Separation between logical servers and physical nodes is a powerful approach, enabling better utilization of resources among different services on a shared computation cluster. If the cluster is split into smaller ones, each of which is dedicated to a particular service, then service A cannot use the idle resources in cluster B even when service B is not actively using cluster B. Deploying both service A and B on the cluster as a whole can alleviate that. Moreover, dynamically reducing/increasing the slave servers running on each cluster node within just a few minutes can maximize the utilization of available resources and also reduce energy consumption in real time.

The execution of popular algorithms like Apriori [4] and FP-Growth [11], even their distributed implementations [6], is row-based, where a transaction row is taken for granted as the execution unit. However, given the proposed storage scheme, this assumption is no longer valid and it is not straightforward to apply/generalize the existing algorithms to accommodate to our storage, due to the fundamental differences in data scanning between row-wise storage and columnar storage. In this section, we present the versions of our algorithms used in SILVERBACK.

D. Two Item-set Algorithm

We first demonstrate the column-oriented algorithm for finding frequent two-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 commercial practice, 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. Our algorithm 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 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: Column-oriented algorithm for finding two frequent 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 two item-sets

```
W' ← {x | x ∈ W, length of x column ≥ α}; O ← {}
for each y ∈ W' do
    U_y ← IDs from y column
    for each x ∈ W' and x ≻ y do
        support_{x,y} ← 0
        bf ← x column's Bloom filter
        for each u ∈ U_y do
            if u in bf then
                support_{x,y} += 1
            end
        end
        if support_{x,y} ≥ α then
            append {x, y} to O
        end
    end
end
return O
```

The equivalence between *intersection of columns* and *union of item-sets* allows us to compute other association mining concepts like *lift*, using the proposed storage and algorithm. This equivalence is best illustrated in single item case, but the same property carries over to general case as shown in [12] and in the following section.

E. Two Issues With Apriori

Two particular operations in the Apriori algorithm significantly slow down its execution time. 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.

1) Minimizing scans of transactions: Apriori algorithm classifies candidate item-sets and explores their candidacy by

Algorithm 2: 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 $H_i(c_2)$
      $SIG(h_i(c_1 \cup c_2)) \leftarrow$ find the smallest m elements from $SIG(h_i(c_1)) \cup SIG(h_i(c_2))$;
      Calculate $J_i\widehat{(c_1, c_2)}$ based on Equation 5
    end
    $J_{hybrid}\widehat{(c_1, c_2)} \leftarrow \sum_{i=1}^{f} \frac{J_i\widehat{(c_1,c_2)}}{f}$
    if $J_{hybrid}\widehat{(c_1, c_2)} \cdot (S_{c1} + S_{c2}) \geq \alpha$ **then**
      $c \leftarrow c_1 \cup c_2$
      order elements in c
      append c to C_k
    end
  end
end
return C_k

Algorithm 3: 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

$O \leftarrow \{\}$
$F_1 \leftarrow \{x | x \in W,$ and $support_x \geq \alpha\}$
$F_2 \leftarrow$ Algorithm1$(\alpha, W, \mathcal{D}_U)$
$O \leftarrow O \cup F_1 \cup F_2$; $k \leftarrow 2$
for *each* $c \in F_2$ **do**
  $S_c \leftarrow$ support counts from Algorithm1's byproduct
  $H_1(c), \ldots, H_f(c) \leftarrow$ obtained from Algorithm1
end
while $F_k \neq \emptyset$ **do**
  $k \mathrel{+}= 1$
  $C_k \leftarrow$ apriori-gen$(F_{k-1}, \alpha,$ $\{H_1(c), \ldots, H_f(c), support_c,$ for $\forall c \in F_{k-1}\})$
  order elements in C_k
  for *each* $c \in C_k$ **do**
    $H_1(c), \ldots, H_f(c) \leftarrow$ empty ascending priority queues each with capped capacity m
    $support_c \leftarrow 0$; $bf \leftarrow$ vector of 1s
    $y \leftarrow$first item in c; $U_y \leftarrow$ IDs from y column
    for *each* $x \in c \backslash y$ **do**
      $bf \leftarrow$ AND-mask$(bf, x$ column Bloom filter$)$
    end
    for *each* $u \in U_y$ **do**
      $h_1, \ldots, h_f \leftarrow u$'s indices in bf, respectively
      if $h_1, \ldots, h_f$ *all set in* bf **then**
        $support_c \mathrel{+}= 1$
        append $h_1, \ldots, h_f$ to $H_1(c), \ldots, H_f(c)$, respectively
      end
    end
    if $support_c \geq \alpha$ **then**
      append c to F_k; append c to O
    end
  end
end
return O

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 [11] offers improvements partially due to the fact that it only scans the database of transactions twice in building the FP-tree structure. However, as mentioned in Section III, 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 [12] 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.

2) Candidate Generation and Probabilistic Pruning: Traditionally, avoiding the exponential growth of candidate item-sets ($2^{|W|}$ possible candidates) by the Apriori principle and other algorithmic improvements [10], was based on pruning the unqualified candidate item-sets. Apriori principle becomes especially effective when $\mathcal{D}_U$ is sparse and contains large number of items and transactions, which exactly suits our practical usage.

The *Apriori-gen* function in Algorithm 3 uses $F_{k-1} \times F_{k-1}$ method [26] to generate, C_k, the set of candidates for frequent k-item-sets. *Apriori-gen* function then uses a new, minHash-based [27] 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 [26]. 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:

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

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 the 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. But we think one can alternatively ask, based on the $F_{k-1} \times F_{k-1}$ method, *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:

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

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:

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

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

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

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 Equation 4 is

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

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 Equation 5 basically as costly as Equation 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:

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

by replacing k in Equation 4 with f. A potential problem with this scheme is that, to achieve reasonable accuracies 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 i-th Bloom hash function. Then the i-th 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 Equation 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 $\widehat{J_{hybrid}(c_1, c_2)}$ is derived as

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

In fact, Equation 6 is a special case of the hybrid estimator. When $k = f$ and $m = 1$, Equation 7 becomes equivalent to Equation 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 if $|S(c)| \geq \alpha$, then $J(c_1, c_2) \cdot (|S(c_1)| + |S(c_2)|) \geq \alpha$, where α is the min support. Replacing $J(c_1, c_2)$ with $J(\widehat{c_1, c_2})$ 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.

F. The SILVERBACK Algorithm

The general association mining algorithm with the proposed pruning technique is presented in Algorithm 3. 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 3, the only temporal performance impact is line 26, where the hash indices (which we get 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 capped length m. The temporal cost for each ID in the test of each candidate without insertions to priority 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.

f, the number of Bloom hash functions, 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.

V. EXPERIMENTAL OBSERVATION

We now present the experiments that we conducted for evaluating the proposed methodologies.

A. Dataset

Our data is collected from two widely used social media platforms: Facebook and Twitter. Both Facebook and Twitter are a medium for individuals, groups or businesses to post content such messages, promotions or campaigns. The user comments/tweets, and user information from specific *interests* is publicly available and collected using Facebook API[2] and Twitter API[3]. In the experiments, the data collected over 2012 is used. Table II shows the size of the databases we are maintaining using the proposed infrastructure and the amount of data used in the experiments.

TABLE II
DATASETS SUMMARY STATISTICS

Statistic	Facebook	Twitter
Unique items/interests	32K+	11K+
(used in experiments)	22,576	4,291
Total user activities	10B+	900M+
(used in experiments)	226M	24.2M
Unique users/transactions	740M+	120M+
(used in experiments)	27.4M	3.7M

B. 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. 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 III
COMMON USER COUNT

interest	TM	CM	C1	C2	C3	C4	C5	C6	C7
EASPORTS	242399	1647	33197	1647	10085	6611	1708	2136	1714
techcrunch	202812	12295	32579	12295	17105	15647	12950	13147	12496
iTunesMusic	189565	7265	24171	7265	10625	9698	7513	7640	7640
google	149577	12022	21352	12022	13797	13621	12605	12636	12636
facebook	120724	8904	14212	8904	9746	9859	9356	9365	9365

[2]http://developers.facebook.com/
[3]https://dev.twitter.com/docs/

TABLE IV
FALSE POSITIVES

interest	C1	C2	C3	C4	C5	C6	C7
EASPORTS	31550	1402	8438	4964	61	489	67
techcrunch	20284	1085	4810	3352	655	852	201
iTunesMusic	16906	568	3360	2433	248	375	375
google	9330	648	1775	1599	583	614	614
facebook	5308	469	842	955	452	461	461

TABLE V
ACCURACY

interest	C1	C2	C3	C4	C5	C6	C7
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

TABLE VI
PRECISION

interest	C1	C2	C3	C4	C5	C6	C7
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

TABLE VII
F-MEASURE

interest	C1	C2	C3	C4	C5	C6	C7
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 III shows 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 IV 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 (table not shown) are easily deduced. The accuracy, precision and F-measure is provided in Table V, VI and VII, respectively.

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 100K vs. 200K 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 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*, are 61 and 67 respectively. But the false positives drop for *techcrunch* when sampling is used.

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 [29] to evaluate our results. Measuring the rank difference instead of absolute error that our probabilistic algorithm makes is due to practical interests. It is more often the case that our customers would ask queries like the *top X number of frequent items* associated with my brand. τ is defined as the ratio of the difference between 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 VIII 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 2-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.

TABLE VIII
KENDALL τ RANK CORRELATION TABLE

Measure	200K, 0.02	200K, 0.002
Kendall τ-statistic	0.98251	0.98455
2-sided p-value	< 0.00001	< 0.00001
S, Kendall Score	3847	3855
Var (S)	79624.33	79624.34
S/τ, Denominator	3915.5	3915.5

C. Temporal Scalability and Efficiency

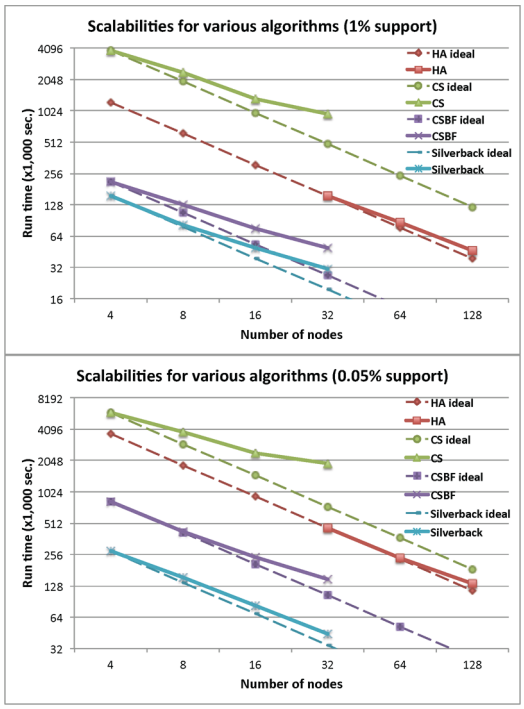

Fig. 3. Scalability comparison

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 Figure 3, we report the run times for different combinations of computing nodes, and minimum support threshold values, for four different algorithms. In the legend of Figure 3, HA denotes the naive implementation of Apriori in the MapReduce framework [5]. 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 Figure 3.

In both support levels (0.05% & 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 run time 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.

VI. CONCLUSIONS

We presented the SILVERBACK framework, a novel solution for association mining from a very large database under constraints of a 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. Our Apriori-based mining algorithm 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 run time, while our probabilistic approach yields a satisfactory level of accuracy.

The SILVERBACK framework has been successfully deployed and maintained at Voxsup since May 2011. Our ongoing efforts are focusing on further improvement our system performance and scalability. Specifically, in the near future we would like to develop more efficient inter-nodal communication solutions, which is critical to scale to hundreds of nodes.

ACKNOWLEDGMENTS

This research was supported by Voxsup, Inc. and in part by NSF awards CCF-0833131, CNS-0830927, CNS-0910952, III-1213038, IIS-0905205, CCF-0938000, CCF-1029166, ACI-1144061, and IIS-1343639; DOE awards DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309,

DESC0005340, and DESC0007456; AFOSR award FA9550-12-1-0458. We are thankful to our colleagues Zhengzhang Chen and Yu Cheng for their insights in our technical discussion.

REFERENCES

[1] Y. Xie, Z. Chen, K. Zhang, M. M. A. Patwary, Y. Cheng, H. Liu, A. Agrawal, and A. N. Choudhary, "Graphical modeling of macro behavioral targeting in social networks." in *SDM*. SIAM, 2013, pp. 740–748.

[2] Y. Xie, Y. Cheng, D. Honbo, K. Zhang, A. Agrawal, A. N. Choudhary, Y. Gao, and J. Gou, "Probabilistic macro behavioral targeting," in *DUBMMSM*, 2012, pp. 7–10.

[3] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD '93*. ACM, 1993, pp. 207–216.

[4] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. VLDB Endow.*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499.

[5] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce," in *ICUIMC '12*. ACM, 2012, pp. 76:1–76:8.

[6] Y. Ye and C.-C. Chiang, "A parallel apriori algorithm for frequent itemsets mining," in *SERA '06*. IEEE, 2006, pp. 87–94.

[7] S. Chung and C. Luo, "Parallel mining of maximal frequent itemsets from databases," in *ICTAI '03*, 2003, pp. 134–139.

[8] M. J. Zaki, S. Parthasarathy, and W. Li, "A localized algorithm for parallel association mining," in *SPAA '97*. ACM, 1997, pp. 321–330.

[9] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Chang, "Pfp: parallel fp-growth for query recommendation," in *RecSys '08*. ACM, 2008, pp. 107–114.

[10] R. J. Bayardo, Jr., "Efficiently mining long patterns from databases," in *SIGMOD '98*. New York, NY, USA: ACM, 1998, pp. 85–93.

[11] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *SIGMOD '00*. New York, NY, USA: ACM, 2000, pp. 1–12.

[12] M. J. Zaki, "Scalable algorithms for association mining," *IEEE TKDE*, vol. 12, no. 3, pp. 372–390, May 2000.

[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *OSDI '06*. USENIX Association, 2006, pp. 15–15.

[15] L. Wu, R. Sumbaly, C. Riccomini, G. Koo, H. J. Kim, J. Kreps, and S. Shah, "Avatara: Olap for web-scale analytics products," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1874–1877, Aug. 2012.

[16] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan, "Distributed cube materialization on holistic measures," in *ICDE'11*, 2011, pp. 183–194.

[17] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[18] B. Lan, B. C. Ooi, and K.-L. Tan, "Efficient indexing structures for mining frequent patterns," in *ICDE '02*. IEEE Computer Society, 2002, pp. 453–462.

[19] Sparse matrices, http://docs.scipy.org/doc/scipy/reference/sparse.html.

[20] Mongodb, http://www.mongodb.org.

[21] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: a high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[22] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985.

[23] Mysql, http://www.mysql.com.

[24] Tornado, http://www.tornadoweb.org/en/stable/.

[25] Facebook's scribe technology, http://www.facebook.com/note.php?note_id=32008268919.

[26] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, 1st ed. Addison Wesley, May 2005.

[27] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang, "Finding interesting associations without support pruning," *IEEE TKDE*, vol. 13, no. 1, pp. 64–78, 2001.

[28] R. Turrisi and J. Jaccard, *Interaction effects in multiple regression*. Sage Publications, Incorporated, 2003, vol. 72.

[29] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1/2, pp. pp. 81–93, 1938. [Online]. Available: http://www.jstor.org/stable/2332226