Technical Report
Number: NU-EECS-13-06

July, 2013

Silverback: Scalable Association Mining For Massive Temporal Data in Columnar Probabilistic Databases

Yusheng Xie, Diana Palsetia, Kunpeng Zhang, Ankit Agrawal, Goce Trajcevski, Alok Choudhary

Abstract

We investigate large scale probabilistic association mining on modest hard- ware infrastructure. We first propose a probabilistic columnar infrastructure for storing the transaction database. Using Bloom filters and reservoir sampling techniques, the storage is efficient and probabilistic. Then we propose an accurate probabilistic algorithm for mining frequent item-sets. Our algorithm relies on the Apriori principle but has a novel probabilistic pruning technique, which reduces frequent item-set candidates without counting every candidate's support size. In the experiments, our Silverback framework, with satisfying accuracy, outperforms Hadoop Apriori implementation in terms of run time. Silverback has been commercially deployed and developed at Voxsup Inc. since May 2011.

Keywords

Association Mining, Frequent pattern mining, consumer behavior, Facebook , Twitter

Silverback: Scalable Association Mining For Massive Temporal Data in Columnar Probabilistic Databases

Yusheng Xie[12], Diana Palsetia[13], Kunpeng Zhang[3], Ankit Agrawal[3], Goce Trajcevski[3], Alok Choudhary[2]
[1]: authors contributed equally,
[2]: Voxsup, Inc., [3]: Northwestern University,
[2]: {yves,alok}@voxsupinc.com,
[3]: {drp925,kzh980,ankitag,goce}@eecs.northwestern.edu

Abstract

We investigate large scale probabilistic association mining on modest hardware infrastructure. We first propose a probabilistic columnar infrastructure for storing the transaction database. Using Bloom filters and reservoir sampling techniques, the storage is efficient and probabilistic. Then we propose an accurate probabilistic algorithm for mining frequent item-sets. Our algorithm relies on the Apriori principle but has a novel probabilistic pruning technique, which reduces frequent item-set candidates without counting every candidate's support size. In the experiments, our Silverback framework, with satisfying accuracy, outperforms Hadoop Apriori implementation in terms of run time. Silverback has been commercially deployed and developed at Voxsup Inc. since May 2011.

1. Introduction

Behavioral targeting refers to techniques used by advertisers whereby they can increase the effectiveness of their campaigns by capturing data generated by user activities. With the advent of social media sites there has been massive growth of user generated content. In the context of social websites behavioral data is generated in the form of likes, posts, retweets, or comments. Behavioral data is foremost characterized by its 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 same set of data mining techniques developed for more traditional data sources. Analyzing the public social web and extracting the most relevant items (i.e. frequent item-sets) for a given commercial interest is a valuable application of association rule mining to large behavioral databases. An *interest* could mean a group of online users, a brand or a product. For example, the brand "Nikon" is a description of an interest in cameras. For a set of given 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. In other words, this operation is analogous to finding frequent item-sets and association rules from a large number of transactions of co-occurrences of the items (6).

But unlike traditional association mining, we work with a more complex and much bigger database in the online behavioral world. In fact, scale is the one factor really making this challenge different from 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 is growing by over 30 million transactions every day. Performing association mining at such scale with modest hardware forces our team to look for an efficient algorithm and a scalable storage scheme alternative to existing methods.

The massive scale of our behavioral database, besides being a formidable challenge, is also an opportunity to fully exploit statistical power. Statistical estimations and probabilistic treatments are key in taking scalability and performance to the next level in both the mining algorithm and database storage, provided that such estimations do not noticeably affect the accuracy of the mining results.

1.1. Our Contribution

To handle association mining from a very large live database with modest infrastructure, we propose Silverback, a probabilistic framework for accurate association rules and frequent item-sets mining at massive scale. First, we propose a column-based storage for storing and updating a large database of transactions. By incorporating Bloom filters and sampling techniques in our storage solution, it results in a much faster probabilistic database with satisfying accuracies. We show that the column-based storage is more efficient and more scalable than traditional row-based databases. Further, we propose an Apriori-based algorithm that efficiently and probabilistically prunes the candidates in constant time without support counting for every candidate item-set. In the experimental section, we compare the performance of Silverback with a more powerful ad-hoc Hadoop cluster; evidences suggest that Silverback is significantly more efficient than a generic MapReduce implementation.

The described framework and algorithm have been successfully deployed at large scale for commercial use and progressively improved to the current version since May 2011.

2. Problem Formulation

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

In technical terms, let $\mathcal{D}$ denote a long list of users' activities across public walls in the Facebook network (or handles from Twitter). $\mathcal{D} = \{(u_i, w_i, t_i, a_i) | i = 0, 1, \ldots, |\mathcal{D}|\}$, where each four-element tuple is a transaction of user activity. For the 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, $|U| << |\mathcal{D}|$ and $|W| << |U|$.

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

A frequent item-set F, given minimal support level α, is a subset of W such that there are at least α number of 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. $\mathcal{D}_U$ in our challenge is equivalent to an 800-million-by-30,000 table that would have over 20 *trillion* cells in full representation.

3. Related Work

Table 1: Comparison with popular association mining algorithms

Algorithm	**Trnsctn. storage**	**Freq. Itms. Reprsntn.**	**Db. scans**	**Memory Footprint**	**Cluster scalability**	**Empirical efficiency**	**Support count**	**Lines of code**	**Accuracy**
Apriori	Row-based	Row-based	Many	Large	Good(24)	benchmark	Yes	$\sim$1,000	Exact
Max-Miner	Row-based	Row-based	Many	Large	Fair(11)	$\sim 5\times$	Yes	Unknown	Exact
Eclat	Columnar	Flexible	A few	Small	Poor(26)	$3\times \sim 10\times$	Yes	$\sim$2,000	Exact
FPGrowth	Row-based	FP tree	2	Enormous	Very Good(17)	$5\times \sim 10\times$	Yes	7,000+	Exact
SILVERBACK	Columnar	Flexible	A few	Tiny	Good	$> 15\times$	Const. time	$\sim$2,000	Probabilist

3.1. Association Mining

Association Mining is a well known technique for finding correlations between items in a dataset. Despite recent advances in parallel association mining algorithms (24) (18), the core technique is largely unmodified. Apriori (7) is the most popular and successful algorithm. It proceeds by identifying the frequent items by starting with small item-sets, and only proceeding to larger item-sets if all subsets are frequent. However, Apriori is very expensive because in every count step it scans the entire database. Many 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. We discuss Max-Miner in the context of generating and pruning candidates, FP growth and Eclat in the context of counting strategy, scanning, and representing database.

3.1.1. Max-Miner

Max-Miner (8) 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. It is able to reduce 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, and therefore 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 search time, it still needs many passes of the transactions to get all long patterns just like basic Apriori. It is 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.

3.1.2. FP Growth

FP-Growth allows frequent item-set discovery without candidate item-set generation. This makes it faster than Apriori. FP-Growth approach builds a compact data structure called the FP-tree. The FP-tree can be constructed by allowing two passes over the data-set. Traversing through the FP-tree allows frequent item-sets to be discovered.

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 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). It also adopts 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 the tree may vary and may not fit in memory. Also the support can only be computed once the entire data-set is added to FP-Tree.

3.1.3. Eclat

Like FP-growth, Eclat employs divide and conquer strategy to decompose the original search space (25). It allows frequent item-set discovery via transaction list(tid-list) intersections and is the first algorithm to use column-based representation of the data rather than row-based representation. The support of an item-set is determined by intersecting the transaction lists for two subsets, and the union of these two subsets is the item-set.

The algorithm performs depth-first search on the search space. For each item, it scans the database to build a list of transactions containing that item (step 1). 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 (step 2). It then repeats step 1 on item-conditional database. This process is repeated for all other items as well.

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

3.1.4. 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 (26), Parallel Max-miner (11), Parallel FP growth (17) and Distributed Apriori (24).

Table 1 compares our proposition with other popular existing methods in many aspects including their scalability to more nodes.

3.2. 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. A bloom filter is popular space-efficient probabilistic data structure used to test membership of an element (9). 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 (10). Similar to Google's BigTable, Apache modeled HBase, which is a Hadoop database. HBase employs bloom filters for few different use-cases. One is 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 a fixed constant, $O(k)$ where k is the number of hash functions, and therefore independent of the number of items already in the set. The only caveat 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: $m = -n \ln p/(\ln 2)^2$.

3.3. OLAP and Data Warehouse

Traditional OLAP (On-Line Analytical Processing) queries are usually generated by aggregation along different spatial and temporal dimensions at various granularity. For example, OLAP queries for a typical job posting website (23) include *job views by day/week/month*, *job views by city*, and *job views by company.* To achieve efficient OLAP queries, queries are issued against specifically designed data warehouses. Current industrial practices usually build a data warehouse from three steps. The first step is cubifying of the raw data. The second step is storing the cubes. And the third step is mapping OLAP queries into cube-level computations.

Web-scale real-time applications like Twitter and Facebook pose tremendous challenges to the three simple 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 *river*. A river is a persistent link between source and destination carrying real-time data stream. In addition, powerful and elastic MapReduce frameworks like Hadoop (19) are usually deployed to handle the first step, the cubification of web-scale data, and the third step, mapping queries to cube-level computations. Durable and scalable key/value pair storages like Cassandra (16) 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.

What if there isn't more hardware or if a Big Data startup simply could't afford sufficient computing power? The pressure is not as much for Facebook's or Twitter's in-house engineers compared to the ones with their much-smaller-in-scale partner startups. Many such startup companies have one or two database engineers to design a system that would need to handle basically inundating amount of data sent from their larger social network partners.

Constraints on the budget and even considerations for energy-saving computing call for design alternative to simply "put it on more machines and scale". The priority, for the smaller entities/companies, should not be inventing general-purpose, idealistic scalable framework for exa-scale computing. The goal for smaller entities/companies is immediate practicality with reasonable scalability on commodity hardware.

4. Storage and Infrastructure

Given the scale of $\mathcal{D}_U$ in our problem, traditional row-based storage assumed by (7) (13) would become out of depth. An efficient and clever storage scheme is the foundation of performance and scalability. Our goal here is not to invent a general-purpose advanced distributed storage engine to add to the already abundant list of such engines and file systems. Instead, we are more interested in an application/data-driven ad-hoc solution. We discover that a probabilistic column storage is very effective for us to tackle the massive scale.

4.1. Scalable Column Storage

One key observation taken into consideration in our design of the storage is the sparsity of $\mathcal{D}_U$. The full representation of $\mathcal{D}_U$ would require over

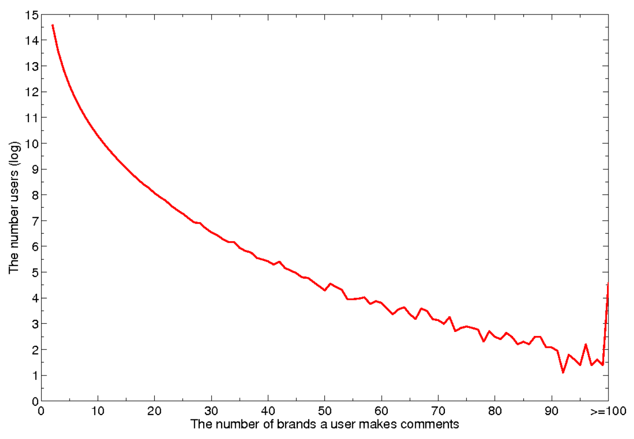

Figure 1: Facebook user activity distribution (2008 June to 2012 January, vertical axis is in log scale)

20 trillion cells (740M users by 32K walls), which is impractical even in distributed environment, for any reasonable budget. But of the 20 trillion cells, less than 1% are populated. That is, for an average user, he/she accesses less than 14 of the 32K walls, according to our estimation. The sparsity of $\mathcal{D}_U$ is not a coincidence and does not surprise us at all. 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. Figure 1 shows the distribution of Facebook users and the number of walls (items) they access. It shows that the number of users accessing x number of walls drastically decrease as x increases. Over 40% of the users only access less than 5 of the 32,000 walls. The spike on the right side is due to aggregating all users with more than 100 accessed walls into a single category. Figure 1 implies that most transactions in $\mathcal{D}_U$ only contain a small number of items.

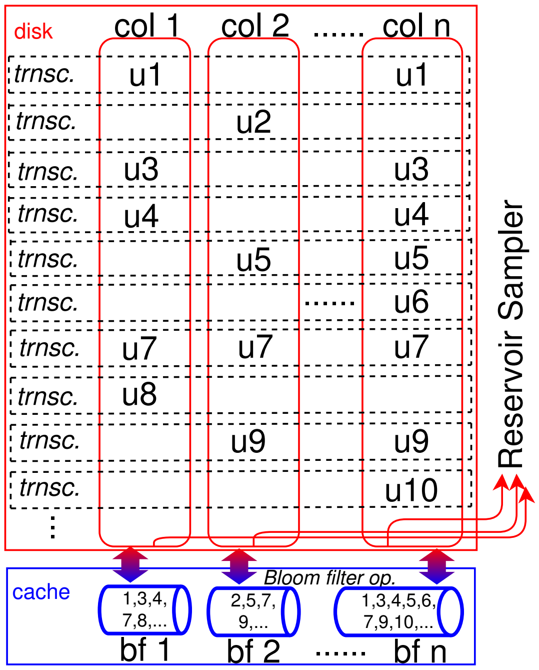

Figure 2: Illustration of the columnar storage in place of traditional row-based transactions and its probabilistic enhancement

We use a sparse representation of the massive $\mathcal{D}_U$ called "list of lists" (LIL) (4) (or "Column Family" in Cassandra (16)). LIL typically stores a massive sparse matrix by using a list to record the non-zero cells for each row. But unlike most existing implementations of LIL, 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; 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.

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 $\mathcal{D}_U$.

4.2. Probabilistic Enhancement

An important characteristic about the LIL representation of $\mathcal{D}_U$ is that the lists/columns for the walls will have drastically different lengths due to the sparsity in $\mathcal{D}_U$. The wall-list for Coca-Cola on Facebook contains over 30 million user IDs while small (albeit important) interests like ACM SIGMOD have less than 100 user IDs in their lists.

An immediate problem caused by the massive size differences is resource allocation. The resource allocator would face a combinatorial problem, where 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 solve this problem, two approaches seem natural. One way is to shard the longer columns like Coca-Cola. But this approach 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 (2). 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.

Instead, our philosophy is simple: solve the locking problem by avoiding it. Like (14), we impose each column file to be single-threaded and therefore, no lock mechanism or extra complex management is needed at all. The trade-off here is the need to make sure each column file size can be handled by a single thread in reasonable delay. Sampling can alleviate the size difference among columns and make large columns controllable by a single-thread. 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 the 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 column files still cannot fit into the main memory of our modest cluster, even after sampling on large columns. Loading all column files of the described $\mathcal{D}_U$ requires roughly 300GB after sampling. The practical goal here is to reduce the representation of $\mathcal{D}_U$ from 300GB down to about 25GB, which needs to be done without breaking data independency, performance or scalability. With such constraints, our options are limited. Sampling based techniques cannot be used since any sampling would have happened in the previous stage; coding-based information compression is also undesirable because of its impact on performance and updatability.

Bloom filter, in the given scenario, seems a legitimate choice. A Bloom filter is a space-efficient probabilistic data storage (9). The idea here 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 shrinking 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, and doing the same for all 30,000 columns would yield 854.5 KBytes $\cdot 30,000 <$ 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. And suppose that each ID sets 7 different bits in the filter, at most 50% of the bits in the Bloom filter will be set, which guarantees the false positive rate on the filters.

Together, the sampling limit and the size of the filter guarantees decent accuracy. This equal-in-size requirement might seem unnecessary and superfluous now, but it is imposed to enable bit operations between any two Bloom filters, which is critical in our association mining algorithm.

One can not get away with using sampling and Bloom filter without discussing their actual impact on the accuracy. In the experimental section, concrete evidences are provided to support the case. In short, both sampling and Bloom filter have very limited impact on the accuracy of the results.

4.3. 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 (3) and MongoDB (2) 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, which are described in Section 5. Algorithms are implemented as web services and are served from scalable web servers like Tornado (5). 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. But physically, several slave servers can reside on a same actual node; and 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, first, implementing the computation as web service-based transactions and, second, 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 precisely put, 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 touching 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 (1). Another advantage particularly interesting for our commercial application is web-servers' builtin handling for timeout requests. Suppose the system is calculating frequent item-sets on-the-fly from end clients' requests. The expectation is not finding complete and exact frequent item-sets in as little time as possible. Instead, the clients expect to explore as many frequent item-sets as possible after a tolerably short delay, say, 1 second. Service-based implementation makes such expectations easy to achieve.

Separation between logical servers and physical nodes is a very powerful idea on a shared computation cluster. The motivation of sharing a bigger cluster among several different services is to allow better utilization of resources. 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. To avoid such inefficiency, both service A and B are deployed on the whole cluster. 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.

5. Algorithm

New algorithms are designed to work with the proposed probabilistic database.

Popular and successful algorithms like Apriori (7) and FP-Growth (13), even their distributed implementations (24), proceed by row-based execution. A transaction row is taken for granted as the execution unit. However, given

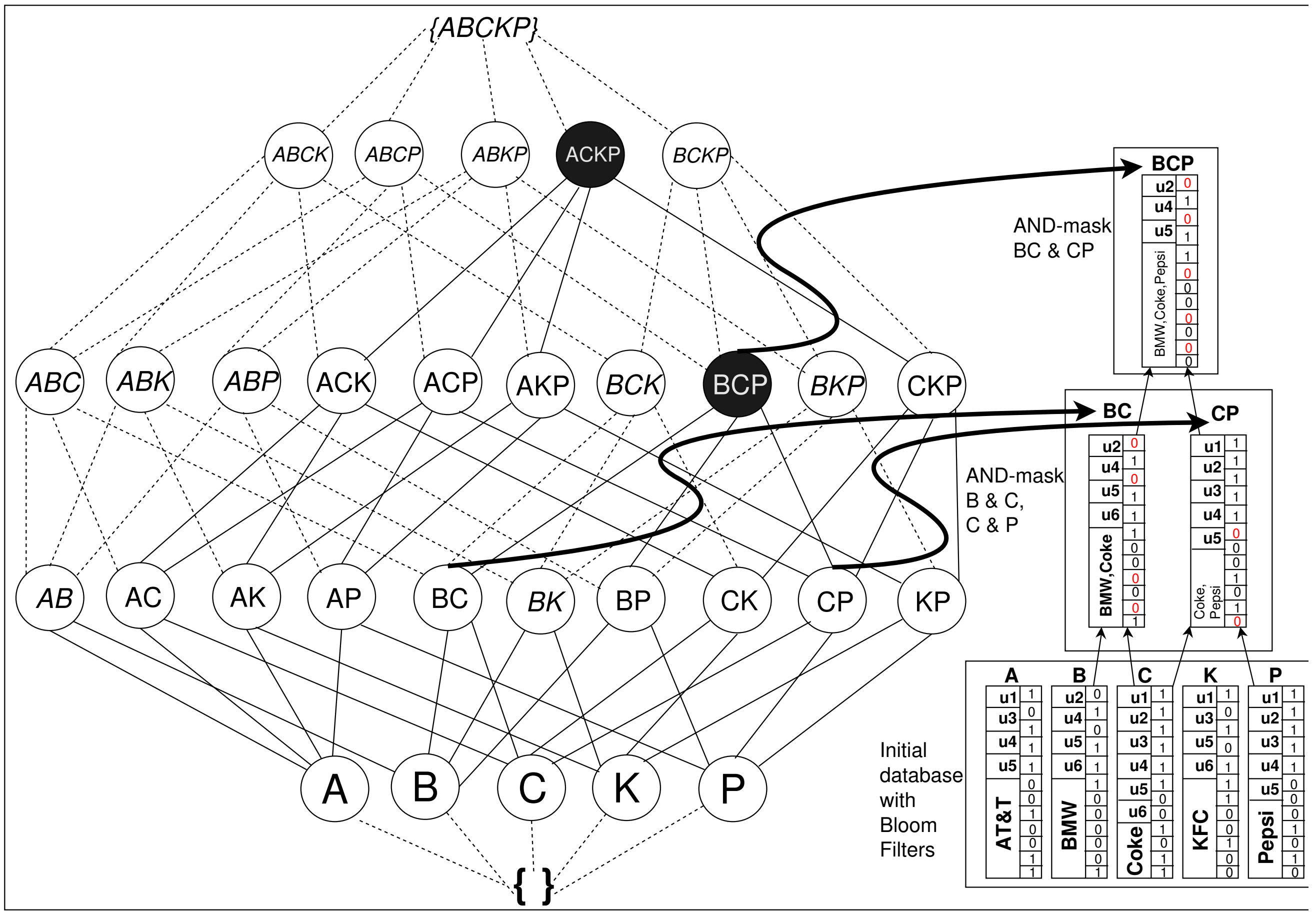

Figure 3: Correspondence between SILVERBACK and lattice representation of frequent item-sets

the proposed storage scheme, this assumption is no longer valid and existing algorithms are hard to generalize to accommodate our storage due to the fundamental differences in data scanning between row-wise storage and columnar storage.

5.1. Two Item-set Algorithm

We first demonstrate the column-oriented algorithm in the case of finding frequent two-item-sets $\{X = \{x\}, Y = \{y\}\}$, where X and Y are both single item-sets, with 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 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. This process can be done very efficiently by scanning $O(|W| - 1)$ numbers, since the algorithm simply queries the length of each column file.

We use W' to denote a subset of W such that W' contains all 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 in Bloom filter and sit on a much slower disk, reading user IDs from disk only happens once per wall to avoid cost, and U_y at each iteration is small enough to fit in memory. In other words, the algorithm scans the whole database from disk only once. Then for each wall's Bloom filter representation b_x, where $x \in W'$, the algorithm tests if 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.

Since $y \cap x$ is the set of user IDs that appear in both y column and x column, $y \cap x$ is equivalent to $\{X \cup Y\}$, the rows of transactions that contain both y and x. 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 (25) and in the following section.

5.2. Two Issues With Apriori

We first review the Apriori principle and the key steps in the original Apriori algorithm, which would help to understand how our version of Apriori can further exploit the Apriori principle with column-based sparse database and Bloom filters.

Apriori Principle: *If an item-set is frequent then all of its subsets are also frequent, or if an item-set is infrequent then all its supersets are infrequent.* Moreover, *maximal frequent item-sets uniquely determine all frequent item-sets* (25). An item-set S is called *maximal frequent* if there does not exist an item-set T such that $S \subset T$ and T is frequent.

Two particular operations in the Apriori algorithm significantly slow the algorithm down. The first is the multiple scans of transactions. The other

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

```
Input: $\alpha$, minimal support, $W$, set of all items, $\mathcal{D}_U$, the database of
       transactions
Output: $O$, set of all frequent two item-sets
1  $W' \leftarrow \{x | x \in W, \text{length of } x \text{ column} \geq \alpha\}$; $O \leftarrow \{\}$
2  for each $y \in W'$ do
3      $U_y \leftarrow$ IDs from $y$ column
4      for each $x \in W'$ and $x \succ y$ do
5          $support_{x,y} \leftarrow 0$
6          $bf \leftarrow x$ column's Bloom filter
7          for each $u \in U_y$ do
8              if $u$ in $bf$ then
9                  $support_{x,y} += 1$
10             end
11         end
12         if $support_{x,y} \geq \alpha$ then
13             append $\{x, y\}$ to $O$
14         end
15     end
16 end
17 return $O$
```

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 are proposed, as discussed in the related work section.

Algorithm 2: apriori-gen algorithm for generating and probabilistically pruning candidates

```
Input: $F_{k-1}$, frequent $(k-1)$ item-sets; $\alpha$, 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
1  $C_k \leftarrow \{\}$
2  for $c_1, c_2 \in F_{k-1} \times F_{k-1}$ do
3      if $c_1$ and $c_2$ satisfy Equation 1 then
4          for $i \in \{1, \ldots, f\}$ do
5              $SIG(h_i(c_1)) \leftarrow$ first $m$ indices in $H_i(c_1)$
6              $SIG(h_i(c_2)) \leftarrow$ first $m$ indices in $H_i(c_2)$
7              $SIG(h_i(c_1 \cup c_2)) \leftarrow$ find the smallest $m$ elements from
               $SIG(h_i(c_1)) \cup SIG(h_i(c_2))$;
8              Calculate $\widehat{J_i(c_1, c_2)}$ based on Equation 5
9          end
10         $\widehat{J_{hybrid}(c_1, c_2)} \leftarrow \sum_{i=1}^{f} \frac{\widehat{J_i(c_1,c_2)}}{f}$
11         if $\widehat{J_{hybrid}(c_1, c_2)} \cdot (S_{c1} + S_{c2}) \geq \alpha$ then
12             $c \leftarrow c_1 \cup c_2$
13             order elements in $c$
14             append $c$ to $C_k$
15         end
16     end
17 end
18 return $C_k$
```

5.3. Minimizing scans of transactions

Apriori algorithm classifies candidate item-sets and explores their candidacy by the cardinality of the item-set. And 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 enormous databases. Previously, FP-Growth (13) is successful 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. In other words, FP-Growth algorithm avoids intensive loads on database scans by shifting the load to its own data structure, which is more concise. Eclat (25) finds another way to avoid excessive database scans without creating extra self-defined data structures. Benefiting from its columnar storage, Eclat (25) 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.

5.4. Candidate Generation and Probabilistic Pruning

Traditional approaches to address the issue of exponential candidate item-sets was by the Apriori principle and other algorithmic improvements (8), which prune the unqualified candidate item-sets and prevent the algorithm from exploring all $2^{|W|}$ possible candidates. 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 (20) to generate, C_k, the set of candidates for frequent k-item-sets. *apriori-gen* function then uses a new, minHash-based (12) 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 (20). 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:

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

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 (21) in the *apriori-gen* function:

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

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

MinHash scheme is a way to estimate $J\left(c_1, c_2\right)$ 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:

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

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 an unbiased estimator of $J(c_1, c_2)$ is

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

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 Equation 5 basically as costly as Equation 2. The key step that makes min-Hash 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:

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

by replacing k in Equation 4 with j. A potential problem with this scheme is that, to achieve reasonable accuracies in Bloom filter and minHash, the expectations on j 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

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

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 $\widehat{J\,(c_1, c_2)}$ 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.

5.5. 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.

6. Experiments and Results

6.1. 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[1] and Twitter API[2]. 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.

6.2. 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. Recall that target drive rule is given as $X \Rightarrow Y$ where W is the set of all interest IDs, $X, Y \subset W$, $X \cap Y = \emptyset$, and Y is given as the target. 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.

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

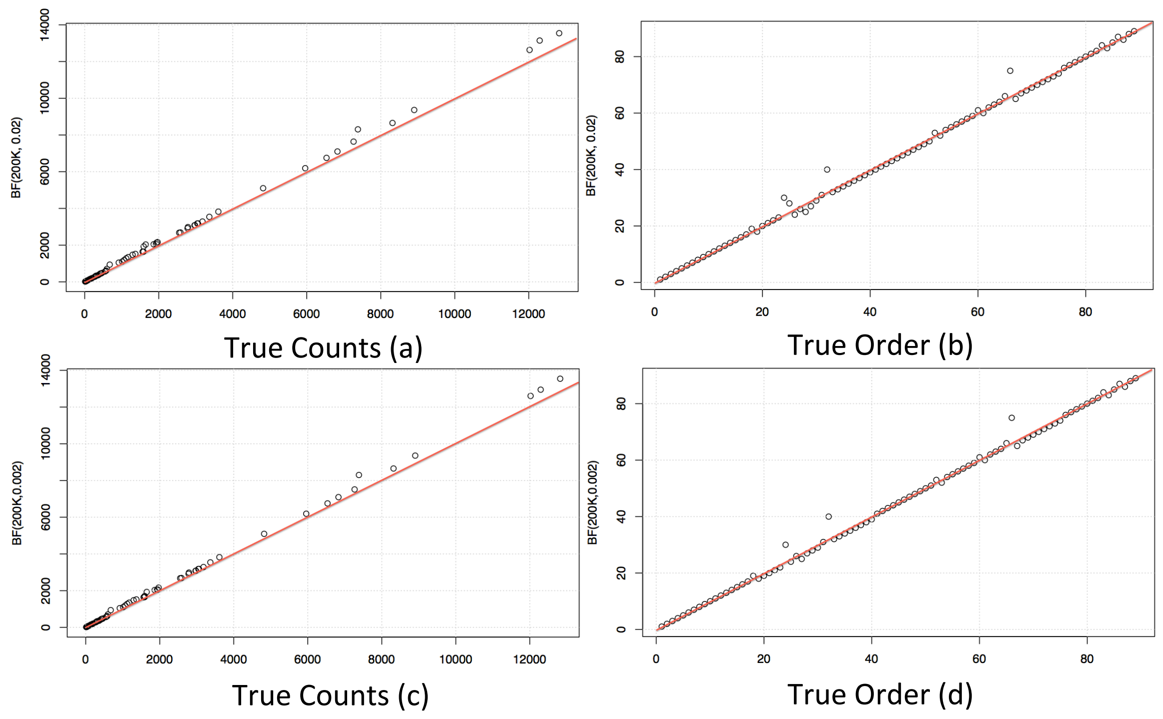

Figure 4: Bloom filter errors visualized as deviations from the red line

Table 3 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 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 (table not shown) are easily deduced. Table 5 provides the number of true negatives. The accuracy, precision and f-measure is provided in Table 6, 7 and 8 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 *200K, 0.002* and *200K, 0.002,S*, are 61and 67 respectively. But the false positives drop for *techcrunch* when sampling is used.

However, note that there can be times when are sampling could result in zero common mentions. This situation does not occur with the shown experiments but should deserve special attention when apply Silverback in practice. A well-chosen sampling ratio can find balance between minimizing such cases and providing good performance without traversing through excessive number of records.

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 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 9 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 certainty. Figure 4 (a) and (c) show how the cardinalities of the support sets of frequent item-sets produced by our probabilistic algorithm differ from what would be produced by an exact mining algorithm. Most dots are above the red line, which reflects the false positive counts introduced by Bloom filters. Figure 4 (b) and (d), on the other hand, show how the rankings of frequent item-sets produced by our probabilistic algorithm differ from what would be produced by an exact mining algorithm. A dot above/below the red line means our probabilistic algorithm overestimates/underestimates its ranking among the frequencies of all frequent item-sets. As we see from Figure 4 and Table 9, the pervasive false positive counts introduced by Bloom filters do not vastly alter the eventual ranking.

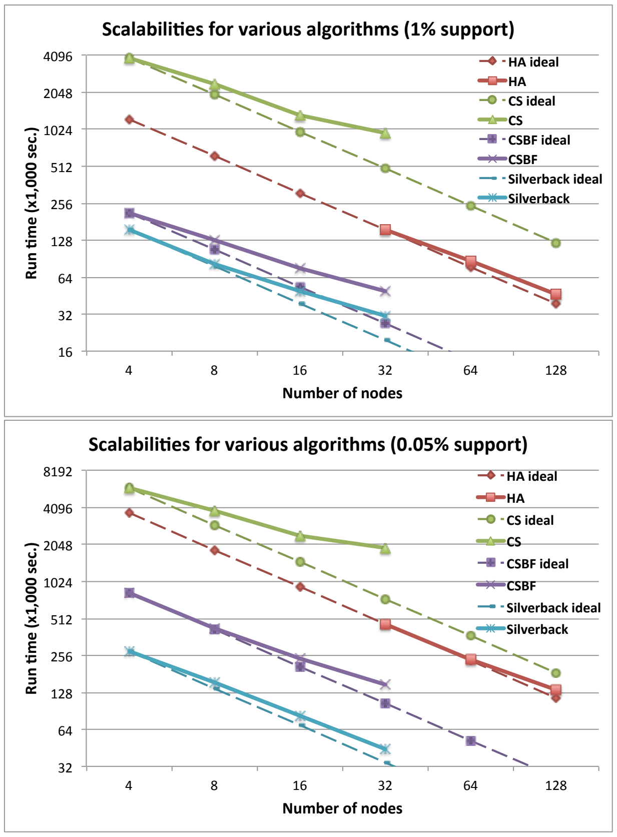

Figure 5: Scalability comparison

6.3. Temporal Scalability and Efficiency

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 5, 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 5, HA denotes the naive implementation of Apriori in the MapReduce framework (18). 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 5.

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 internode 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 level 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.

7. Conclusions

In this paper, we have worked on SILVERBACK, a novel solution for association mining from a very large database under constraints of modest hardware. SILVERBACK is both efficient and scalable. We have proposed accurate probabilistic algorithms for mining frequent item-sets. Our contributions include a columnar storage, which is enhanced by Bloom filters and reservoir sampling techniques, and an Apriori-based mining algorithm, which prunes candidate item-sets without counting every candidate's support. In the experimental section, SILVERBACK outperforms Hadoop Apriori on a more powerful cluster in run time, while our probabilistic approach gives satisfying accuracy.

The SILVERBACK framework has been successfully deployed and maintained at Voxsup since May 2011. In the future, we aim to further improve our system by incorporating a more efficient inter-nodal communication solution, as it would be critical to scale to hundreds or more nodes.

8. Acknowledgments

This research was supported by Voxsup, Inc.

References

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

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

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

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

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

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

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

[8] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *SIGMOD '98*, pages 85–93, New York, NY, USA, 1998. ACM.

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[10] 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*, pages 15–15. USENIX Association, 2006.

[11] S. Chung and C. Luo. Parallel mining of maximal frequent itemsets from databases. In *ICTAI '03*, pages 134–139, 2003.

[12] 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*, 13(1):64–78, 2001.

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate

generation. In *SIGMOD '00*, pages 1–12, New York, NY, USA, 2000. ACM.

[14] 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.*, 1(2):1496–1499, Aug. 2008.

[15] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):pp. 81–93, 1938.

[16] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

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

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

[19] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE'11*, pages 183–194, 2011.

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

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

[22] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, Mar. 1985.

[23] 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.*, 5(12):1874–1877, Aug. 2012.

[24] Y. Ye and C.-C. Chiang. A parallel apriori algorithm for frequent itemsets mining. In *SERA '06*, pages 87–94. IEEE, 2006.

[25] M. J. Zaki. Scalable algorithms for association mining. *IEEE TKDE*, 12(3):372–390, May 2000.

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

Algorithm 3: SILVERBACK - columnar probabilistic algorithm for finding general frequent item-sets

```
Input: $\alpha$, minimal support, $W$, set of all walls, $\mathcal{D}_U$, the database of
        transactions
Output: $O$, set of all frequent item-sets
1  $O \leftarrow \{\}$
2  $F_1 \leftarrow \{x | x \in W,$ and $support_x \geq \alpha\}$
3  $F_2 \leftarrow$ Algorithm1$(\alpha, W, \mathcal{D}_U)$
4  $O \leftarrow O \cup F_1 \cup F_2$; $k \leftarrow 2$
5  for each $c \in F_2$ do
6      $S_c \leftarrow$ support counts from Algorithm1's byproduct
7      $H_1(c), \ldots, H_f(c) \leftarrow$ obtained from Algorithm1
8  end
9  while $F_k \neq \emptyset$ do
10     $k += 1$
11     $C_k \leftarrow$ apriori-gen($F_{k-1}, \alpha,$
12                             $\{H_1(c), \ldots, H_f(c), support_c,$
13                                       for $\forall c \in F_{k-1}\})$
14     order elements in $C_k$
15     for each $c \in C_k$ do
16         $H_1(c), \ldots, H_f(c) \leftarrow$ empty ascending priority queues each with
           capped capacity $m$
17         $support_c \leftarrow 0$; $bf \leftarrow$ vector of 1s
18         $y \leftarrow$first item in $c$; $U_y \leftarrow$ IDs from $y$ column
19         for each $x \in c \backslash y$ do
20             $bf \leftarrow$ AND-mask($bf, x$ column Bloom filter)
21         end
22         for each $u \in U_y$ do
23             $h_1, \ldots, h_f \leftarrow u$'s indices in $bf$, respectively
24             if $h_1, \ldots, h_f$ all set in $bf$ then
25                 $support_c += 1$
26                 append $h_1, \ldots, h_f$ to $H_1(c), \ldots, H_f(c)$, respectively
27             end
28         end
29         if $support_c \geq \alpha$ then
30             append $c$ to $F_k$; append $c$ to $O$
31         end
32     end
33 end
34 return $O$
```

Table 2: Datasets summary statistics

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

Table 3: Common User Count

interest	**TM**	**CM**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
YouTube	1323042	26176	182711	26176	179863	155803	101652	129730	19945
EASPORTS	242399	1647	33197	1647	10085	6611	1708	2136	1714
techcrunch	202812	12295	32579	12295	17105	15647	12950	13147	12496
iTunesMusic	189568	7265	24171	7265	10625	9698	7513	7640	7640
google	149877	12022	21352	12022	13797	13621	12605	12636	12636
facebook	120724	8904	14212	8904	9746	9859	9356	9365	9365
intel	113509	6830	10856	6830	7412	7431	7094	7101	7101
netflix	96357	7383	11754	7383	8562	8657	8302	8308	8308
Xbox	65048	4820	7829	4820	5273	5357	5096	5100	5100
eBay	63478	8317	9229	8317	8674	8700	8654	8654	8654
Microsoft	61996	5959	6703	5959	6208	6227	6189	6189	6189
bing	60261	2978	3529	2978	3097	3118	3080	3080	3080
AppStore	56953	3367	4016	3367	3559	3583	3544	3544	3544
iheartradio	54378	3059	3539	3059	3192	3214	3184	3184	3184
amazonmp3	48312	6534	6955	6534	6752	6765	6751	6751	6751
PlayStation	48286	3184	3516	3184	3290	3312	3289	3289	3289
sprint	47487	2790	3303	2790	2987	3002	2982	2982	2982
XboxSupport	45036	1589	2757	1589	2185	2213	2165	2165	2165
VerizonWireless	35330	3043	3250	3043	3180	3182	3180	3181	3181
nokia	35026	2589	2751	2589	2691	2697	2690	2690	2690

Table 4: False Positives

interest	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
YouTube	156535	149025	153687	129627	75476	103554	-6231
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
intel	4026	276	582	601	264	271	271
netflix	4371	922	1179	1274	919	925	925
Xbox	3009	280	453	537	276	280	280
eBay	912	337	357	383	337	337	337
Microsoft	744	230	249	268	230	230	230
bing	551	102	119	140	102	102	102
AppStore	649	177	192	216	177	177	177
iheartradio	480	125	133	155	125	125	125
amazonmp3	421	217	218	231	217	217	217
PlayStation	332	105	106	128	105	105	105
sprint	513	192	197	212	192	192	192
XboxSupport	1168	576	596	624	576	576	576
VerizonWireless	207	137	137	139	137	138	138
nokia	162	101	102	108	101	101	101

Table 5: True Negatives with Different Bloom Filter Configurations

interest	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
YouTube	1406	149025	4254	28314	82465	155803	101652
EASPORTS	150920	1402	174032	177506	182409	6611	1708
techcrunch	151538	1085	167012	168470	171167	15647	12950
iTunesMusic	159946	568	173492	174419	176604	9698	7513
google	162765	648	170320	170496	171512	13621	12605
facebook	169905	469	174371	174258	174761	9859	9356
intel	173261	276	176705	176686	177023	7431	7094
netflix	172363	922	175555	175460	175815	8657	8302
Xbox	176288	280	178844	178760	179021	5357	5096
eBay	174888	337	175443	175417	175463	8700	8654
Microsoft	177414	230	177909	177890	177928	6227	6189
bing	180588	102	181020	180999	181037	3118	3080
AppStore	180101	177	180558	180534	180573	3583	3544
iheartradio	180578	125	180925	180903	180933	3214	3184
amazonmp3	177162	217	177365	177352	177366	6765	6751
PlayStation	180601	105	180827	180805	180828	3312	3289
sprint	180814	192	181130	181115	181135	3002	2982
XboxSupport	181360	576	181932	181904	181952	2213	2165
VerizonWireless	180867	137	180937	180935	180937	3182	3180
nokia	181366	101	181426	181420	181427	2697	2690

Table 6: Accuracy

interest	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
YouTube	0.150	0.191	0.165	0.296	0.590	0.438	1.312
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
intel	0.978	0.999	0.997	0.997	0.999	0.999	0.999
netflix	0.976	0.995	0.994	0.993	0.995	0.995	0.995
Xbox	0.984	0.998	0.998	0.997	0.999	0.998	0.998
eBay	0.995	0.998	0.998	0.998	0.998	0.998	0.998
Microsoft	0.996	0.999	0.999	0.999	0.999	0.999	0.999
bing	0.997	0.999	0.999	0.999	0.999	0.999	0.999
AppStore	0.996	0.999	0.999	0.999	0.999	0.999	0.999
iheartradio	0.997	0.999	0.999	0.999	0.999	0.999	0.999
amazonmp3	0.998	0.999	0.999	0.999	0.999	0.999	0.999
PlayStation	0.998	0.999	0.999	0.999	0.999	0.999	0.999
sprint	0.997	0.999	0.999	0.999	0.999	0.999	0.999
XboxSupport	0.994	0.997	0.997	0.997	0.997	0.997	0.997
VerizonWireless	0.999	0.999	0.999	0.999	0.999	0.999	0.999
nokia	0.999	0.999	0.999	0.999	0.999	0.999	0.999

Table 7: Precision

interest	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
YouTube	0.143	0.149	0.146	0.168	0.258	0.202	1.312
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
intel	0.629	0.961	0.921	0.919	0.963	0.962	0.962
netflix	0.628	0.889	0.862	0.853	0.889	0.889	0.889
Xbox	0.616	0.945	0.914	0.900	0.946	0.945	0.945
eBay	0.901	0.961	0.959	0.956	0.961	0.961	0.961
Microsoft	0.889	0.963	0.960	0.957	0.963	0.963	0.963
bing	0.844	0.967	0.962	0.955	0.967	0.967	0.967
AppStore	0.838	0.950	0.946	0.940	0.950	0.950	0.950
iheartradio	0.864	0.961	0.958	0.952	0.961	0.961	0.961
amazonmp3	0.939	0.968	0.968	0.966	0.968	0.968	0.968
PlayStation	0.906	0.968	0.968	0.961	0.968	0.968	0.968
sprint	0.845	0.936	0.934	0.929	0.936	0.936	0.936
XboxSupport	0.576	0.734	0.727	0.718	0.734	0.734	0.734
VerizonWireless	0.936	0.957	0.957	0.956	0.957	0.957	0.957
nokia	0.941	0.962	0.962	0.960	0.962	0.962	0.962

Table 8: F-measure

interest	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**
YouTube	0.251	0.260	0.254	0.257	0.410	0.316	1.135
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
intel	0.772	0.980	0.959	0.970	0.981	0.975	0.981
netflix	0.772	0.941	0.926	0.934	0.941	0.937	0.941
Xbox	0.762	0.972	0.955	0.963	0.972	0.968	0.972
eBay	0.948	0.980	0.979	0.980	0.980	0.980	0.980
Microsoft	0.941	0.981	0.980	0.980	0.981	0.981	0.981
bing	0.915	0.983	0.980	0.982	0.983	0.982	0.983
AppStore	0.912	0.974	0.972	0.973	0.974	0.974	0.974
iheartradio	0.927	0.980	0.979	0.979	0.980	0.980	0.980
amazonmp3	0.969	0.984	0.984	0.984	0.984	0.984	0.984
PlayStation	0.950	0.984	0.984	0.984	0.984	0.984	0.984
sprint	0.916	0.967	0.966	0.966	0.967	0.967	0.967
XboxSupport	0.731	0.847	0.842	0.844	0.847	0.845	0.847
VerizonWireless	0.967	0.978	0.978	0.978	0.978	0.978	0.978
nokia	0.970	0.981	0.981	0.981	0.981	0.981	0.981

Table 9: 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