Indexing Bipartite Memberships in Web Graphs

Yusheng Xie[1] Zhengzhang Chen[2] Diana Palsetia[1] Ankit Agrawal[1] Alok Choudhary[1]
[1]: Northwestern University, Evanston, IL USA
[2]: NEC Laboratories America, Princeton, NJ USA
[1]: {yxi389,drp925,ankitag,choudhar}@eecs.northwestern.edu [2]: zchen@nec-labs.com

***Abstract*—Massive bipartite graphs are ubiquitous in real world and have important applications in social networks, biological mechanisms, etc. Consider one billion plus people on Facebook making trillions of connections with millions of organizations. Such big social bipartite graphs are often very skewed and unbalanced, on which traditional indexing algorithms do not perform optimally. In this paper, we propose Arowana, a data-driven algorithm for indexing large unbalanced bipartite graphs. Arowana achieves a high-performance efficiency by building an index tree that incorporates the semantic affinity among unbalanced graphs. Arowana uses probabilistic data structures to minimize space overhead and optimize search. In the experiments, we show that Arowana exhibits significant performance improvements and reduces space overhead over traditional indexing techniques.**

I. Introduction

Search is at the heart of acquiring knowledge. Efficient indexing of large amount of documents has become just as critical as storing the documents. The cost of document indexing can be roughly split into two aspects: spatial and temporal. For spatial cost, one is typically concerned with the spatial requirement in the building process and the index size once the building process finishes. In regard to temporal cost, typical measurements include the initial building cost, incremental insert cost, deletion cost, and access cost.

Among the different types of searchable documents, text documents hold a very special place. Not only is text the conventional and the basic form of interaction between human and computer, it is also widely used to represent other searchable information such as large graphs and networks. On the other hand, most social graphs use document-based (e.g. JSON) API to allow apps or clients to interact with their underlying graphs. This convention in API heavily changes the way we access the graphical information. Consider Facebook as an example. Facebook hosts public pages for a large number of brands and pubic figures. Through its Graph API, Facebook provides authenticated apps with log-like stream of user-brand association as shown in Table I. Each line in Table I is basically a statement of association relationships between users and brands. A natural graphical representation of this kind of association is the so-called heterogenous bipartite graph [1]. A bipartite graph G is defined as $G = (U \cup W, E)$ where $U = \{u_i | 1 \leq i \leq |U|\}$, $W = \{w_j | 1 \leq j \leq |W|\}$, and $E \in U \times W$. G is called a heterogenous bipartite graph when its vertices from U and W model physically distinct categories [2] [3] [1]. For example, heterogenous bipartite graph [4] can also model videos and users on Youtube. In our running example of Facebook graph, U represents the user vertices; W represents the brand vertices; E represents connections between users and brands; and T denotes the input log text.

A special class of bipartite graphs, *unbalanced* heterogeneous bipartite graphs (UHBGs), is emerging from web-scale data. For example, Facebook may have over 1 billion active users, but less than 1 million official public pages are registered on Facebook. An unbalanced bipartite structure poses challenges to existing generic indexing schemes as well as opens possibilities for new specific indexing algorithms. How to build an efficient index from T to support operations such as search and boolean queries is the topic of this paper.

TABLE I
Storing social graph as log entries.

Timestamp	Data
1305123654	`/walmart/[u1,u2,u3,u7,u9]`
1306123657	`/coke/[u0,u2,u4,u7,u8]`
1306823552	`/kohls/[u1,u3,u6,u8]`
1307233628	`/coke/[u5,u6,u7,u8,u9]`

From the receiver's stance (e.g., a startup getting stream T from Facebook), accessing graphical knowledge in $(U \cup W, E)$ from T is awkward. For example, one cannot find out all the `userIDs` associated with `walmart` without scanning all lines in T. One can neither quickly find out all `brands` that `u3` has connection with.

A special class of bipartite graphs, *unbalanced* heterogeneous bipartite graphs (UHBGs)[5], is emerging from web-scale data. For example, Facebook may have over 1 billion active users, but less than 1 million official public pages are registered on Facebook. An unbalanced bipartite structure poses challenge to existing generic indexing schemes as well as opens possibilities for new specific indexing algorithms. Investigating Facebook user's interest distribution among public figures and brands would entail building a huge UHBG, where U contains tens of thousands wall nodes and W contains hundreds of millions of user nodes.

A. Major Contributions

We propose Arowana, a novel bipartite indexing algorithm, whose design is driven by characteristics of web scale social graphs and their applications. Arowana achieves a high-performance efficiency by building its index tree that incorporates the semantic affinity among unbalanced graphs. In Arowana, we propose a probabilistic data structure to minimize space overhead and optimize search. The unique building process of an AROWANA index requires a novel community detection algorithm. Further, we analytically show asymptotic bounds for various query costs when using the AROWANA index. The application of Bloom filters is novel as well. Previous studies [6] [7] use Bloom filters to organize sensor-network data. But we have not seen others giving the exact or similar structures like AROWANA, or designating its purpose to bipartite indexing. In experiments, we show AROWANA's superior scalability and competence in building and retrieving queries over B-Trees and Lucene. We also find out that AROWANA's competitiveness is conditional on the large graph size and the semantic meaning of the social graph.

AROWANA has been developed and commercially deployed at a digital marketing firm since December 2011. AROWANA powered search engines serve millions internal queries per day.

II. PROBLEM STATEMENT

Suppose the input log text T contains n lines like in Table 1. Let $\Sigma_W = \{\texttt{walmart}, \texttt{coke}, \ldots\}$ and $\Sigma_U = \{\texttt{u1}, \texttt{u2}, \ldots\}$ denote the brand alphabet and the user alphabet, respectively. The two alphabets simply correspond to the heterogeneous vertex sets in a bipartite graph G. The goal is to construct a spatio-temporal parsimonious index I on T such that bipartite membership queries, which are fast on G but very slow on T, can be efficiently answered using I.

Since one can always construct the bipartite graph G from T, the proposed index I must have spatio-temporal performance advantage over G for it to pay off. More specifically, I should efficiently support the three basic bipartite membership queries: (1) Select1(w): retrieve all unique `userID`s, who access the brand w; (2) Select2(u): retrieve all unique `brand`s, to which userID u is connected; (3) Connect(u,w): return 1 if there exists a line in T such that the line contains both `userID` u and `brand` w; return -1 otherwise.

Theoretically, B-Trees and key-value stores can efficiently answer the above desiderata. But their drawbacks in spatial overhead among existing indexers are very severe in practice (even in distributed settings) and they often use more resources than really necessary. Our experiments, with real datasets and practical hardware, show that the theoretical bounds for existing index-accessing performance are washed away due to (the lack of) caching and thrashing. Table II summarizes the pros/cons and applicabilities of various indexing schemes.

III. RELATED WORK

Various traditional indexing schemes including B-Tree index and Bitmap index can be applied to index the bipartite memberships. But all of them have pros/cons summarized in Table II.

A. B-Tree Index at Massive Scale

The advantage with B-Tree dictionary is clear. Any Select1 operation based on `brand` is guaranteed to be efficient. In addition, this solution supports *dynamic alphabet*, which means that it is not necessary for the indexer to have the knowledge of all possible items in the alphabet Σ. Its support for dynamic alphabet is the primary reason for its popularity in most existing database systems [9]. However, B-Tree dictionary index has a drawback in spatial efficiency, especially at large scale deployment, which eventually hurts its overall performance. Two independent indices need to be built and maintained to support Select1 and Select2 queries. A tree I_W indexes all `brand,row_p` pairs and can only answer all Select1 queries. A separate tree I_U has to be built for all `userID,row_p` pairs to support Select2 queries. I_W and I_U will eventually compete for the same memory space. It is a known issue that B-Tree cannot scale logarithmically with data-size in practice once the data grows larger than the main memory [1], even on a distributed platform.

B. Bitmap Index For Massive Cardinality

A Bitmap index conceptually keeps a binary list l_t for each unique value t in the dataset. And entries in l_t are set to be 1 if and only if the entries in the original data hold value t. Necessary to keep a separate list for each unique value in data, Bitmap indices are thought [10] to be profitable for only dataset with small cardinality such as Boolean values. But Bitmap index lists would look very sparse once the data set becomes large and high in cardinality. Numerous efforts have been committed to the area of compressing Bitmap index [8] and improving its performance on high cardinality sets [11]. However, Bitmap has another drawback. Other than its inadequacy with high cardinality sets, Bitmap cannot efficiently deal with rapidly growing or frequently updated databases [12]. It is still notoriously hard for Bitmap index to efficiently handle a large, growing, frequently updated database nearly as well as B-Tree [8].

C. Neighbor Query on Compressed Graphs

Neighbor query friendly social network compression techniques recently have been proposed in [13]. That work describes algorithms for compressing social graphs and accessing the compressed graphs. Although not stated explicitly, It uses standard B-tree/hash index for vertices and edges. Our work focuses on indexing and be of additional value to the technique proposed in [13].

IV. AROWANA ALGORITHM

Given the drawbacks in existing technologies, it is desirable to design an algorithm to (1) achieve a good indexing compression ratio because smaller memory footprint translates into practical performance; and (2) allow dynamic alphabet for it to be practical for web-scale applications. The AROWANA tree data structure is illustrated in Figure 1 and is explained in context with the implementation of Select1, Select2, and Connect queries in following subsections.

[1] http://www.slideshare.net/daumdna/mongodb-scaling-write-performance

TABLE II
COMPARISON OF DIFFERENT INDEXING TECHNIQUES.

Property	**Theoretical Bound**	**B-Tree**	**Bitmap**	AROWANA
Select1	$O(\lvert\Sigma_U\rvert)$	$O(\lg(\lvert\Sigma_W\rvert) + \lvert\Sigma_U\rvert)$	$O(\lvert\Sigma_U\rvert)$	$O(\lg(\lvert\Sigma_W\rvert) + \lvert\Sigma_U\rvert)$
Select2	$O(\lvert\Sigma_W\rvert)$	$O(\lvert\Sigma_W\rvert)$	$O(\lvert\Sigma_W\rvert)$	$O(\lvert\Sigma_W\rvert)$
Connect	$O(1)$	$\Theta(\lg(\lvert\Sigma_W\rvert \times \lvert\Sigma_U\rvert))$	$O(\lg(\lvert\Sigma_U\rvert + \lvert\Sigma_W\rvert))$	$O(\lg(\lvert\Sigma_W\rvert)$
Index size	$\lg\left(\binom{\lvert\Sigma_U\rvert \times \lvert\Sigma_W\rvert}{\lvert T\rvert}\right)$	$O(\lvert T\rvert)$	$\lvert T\rvert \lg(\lvert\Sigma_U\rvert + \lvert\Sigma_W\rvert)$ bits	$\Theta\left(\lvert\Sigma_W\rvert \lg\left(\binom{\lvert\Sigma_U\rvert}{n\tilde{l}/\lvert\Sigma_W\rvert}\right)\right)$
Build time	$\Theta(\lvert T\rvert)$	$\Theta(\lvert T\rvert)$	Variable [8]	$O(T + \lvert\Sigma_W\rvert)$
Dynamic lexicon	N/A	Yes	Expensive	Yes
Index locality	N/A	Value-based	Hash-based	Semantic affinity

A. Implementing Select1 Queries

Among the three basic bipartite membership queries, select1 query is the easiest one to implement. To support Select1 on the input log text T, we only need to provide prefix matching up to the second "/" delimiter because following the second delimiter is a retrievable list of `userIDs`. Each line in T would be reduced to one indexable word, the `brand`. In other words, indexing T like a sequence of strings, given consideration to the aforementioned two characteristics, would not be like indexing a sequence of single words from a large alphabet. If AROWANA were to directly apply technique like Wavelet trees (or any other fancy indexing algorithm for this matter), the index tree would be 1 in depth and would be basically a hash table, which is fast but provides hardly any compression.

AROWANA handles Select1 queries similar to regular B-Tree index. One difference is that a B-Tree index needs to expand each line in T for each `userID` and then index (as shown later in Algorithm 2). Expanding each line in T is not needed for Select1 query and it costs more space, but this procedure is required if using B-Tree because later when handling Select2 queries, a separate index is needed to be built on the `userID` column. Another difference of AROWANA from B-Tree is the internal nodes in the tree. Unlike regular B-Tree, which automatically generates internal tree nodes, AROWANA explicitly introduces *artificial* parent nodes to each `brand` from Σ_W. Using artificial parent nodes, we can organize the `brands` from Σ_W into hierarchies of clusters/subtrees.

Suppose there are K different levels of artificial nodes and $K \sim \Theta(\lg(|\Sigma_W|))$. For $1 \leq k \leq K$, let Σ_{P_k} be the set of all artificial nodes of level k. $\Sigma_{P_1}, \ldots, \Sigma_{P_K}$ are progressively smaller in cardinality and Σ_{P_K} is basically a singleton set with the root node. Further, notation-wise it is convenient to express Σ_W simply as Σ_{P_0}. Once the artificial nodes are conceived and put to K levels, we can simply link the elements from Σ_W to parent nodes in Σ_{P_1}, link the nodes in Σ_{P_1} to their parents in Σ_{P_2} and so on. How the tree structure is formed (i.e. which child nodes should be linked to which parent nodes?) has no bearing in Select1 queries because each `brand` in Select1 is queried independently. However, forming the optimal tree structure has critical impact to the Select2 queries, which will be explored later.

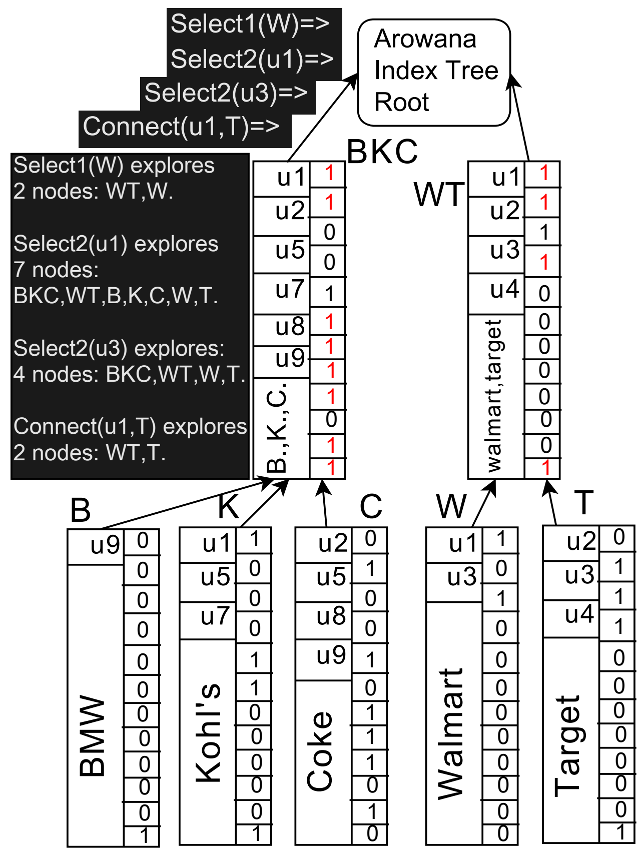


Fig. 1. Each tree node contains Bloom filter structure based on local lexicon. A bit is marked red if it is set during subtree merge. (Best viewed in color)

B. Implementing Connect Queries

Connect membership queries can also be easily supported in the proposed AROWANA tree. Bloom filter is a popular space-efficient probabilistic data structure used to test membership of an element [14]. One straightforward option here is to build one Bloom filter for the set $\{\langle$`brand`, `userID`$\rangle$pairs in $T\}$. However, it is hard to estimate the anticipated inserts to a centralized Bloom filter, which is critical for the filter to retain acceptable false positive rates. Therefore, AROWANA maintains a separate Bloom filter BF_w for each distinct `brand` w that has been observed so far. Note that doing so does not require any knowledge on Σ_W. Maintaining separate Bloom filters have additional advantages such as parallel updating.

Algorithm 1: Reduce Select2 query to Connect queries.

Input: u, any word in Σ_U, Σ_W, alphabet of `brands`
Output: $\Sigma_W(u)$ all `brands` that u is active on

```
Σ_W(u) ← {}
for every w ∈ Σ_W do
    append w to Σ_W(u) if Connect(u, w)
end
return Σ_W(u)
```

The collection of all BF_w for $\forall w \in \Sigma_W$ is sufficient to answer the Connect query efficiently. For Connect(u,w), the algorithm simply tests if u in BF_w.

In an AROWANA tree shown in Figure 1, each artificial parent node also has its own Bloom filter. The Bloom filter of a parent node is the *merge* of all from its children's. The *merge* operation can be as simple as OR operation on the filters (as illustrated in Figure 1). However, simply merging all children's filters by OR will potentially destroy the accuracy of the parent's Bloom filter, which, as a prerequisite of the OR operations, has the same capacity as any of its children's. A simple solution is to employ scalable Bloom filters [14] that can accommodate dynamically increasing capacity of the filter without rehashing the inserted items when expanding the filter. In practice, we recommend using Bit.ly's[2] scalable Bloom filters at the top two levels and fixed-size, fast implementation [3] at lower levels. Even with scalable Bloom filters, the false positive rates at top level nodes are *designed* to be significantly higher than lower level ones. Reinforcing uniform false positive rates at all nodes would cause the AROWANA tree to grow too large in size and outweigh the advantage.

C. Implementing Select2 Queries

So far, the AROWANA index can handle Select1 and Connect membership queries efficiently with minimal cost very close to theoretical lower bound. We still need to show that the AROWANA index handles Select2 queries effectively, which turns out to be nontrivial.

The AROWANA tree shown in Figure 1 seems to support only look ups on `brand` (i.e., Select1, Connect queries). Instead of building another index for Select2 (as in B-Tree or Bitmap), AROWANA can computationally answer Select2 queries just as efficient through optimized searching techniques. The trick is to reduce Select2 queries to a sequence of Connect queries, as illustrated in Algorithm 1.

Algorithm 1 reduces all Select2 queries in the form of Connect queries, which are already efficiently processed by AROWANA. The naive reduction in Algorithm 1 gives a $\Theta(|\Sigma_W|)$ temporal overhead. A more efficient way to translate a Select2 query to Connect queries is to use an AROWANA tree to organize all BF_w for $\forall w \in \Sigma_W$. As illustrated in Figure 1, the five BFs (B, K, C, W, and T) are leaves in the tree. They are adopted by BKC and WT, two artificial parent nodes from Σ_{P_1}. An artificial parent node in Figure 1 is more than a symbol because it holds a structure that represents merged Bloom filters from all its children. Define the associated `userID` set of any node p, $\mathcal{A}_p$, in a AROWANA tree to be the set of all `userID`s such that Connect(p,`userID`) returns positive. Clearly, an artificial parent node can answer a Connect query as to whether the given `userID` is connected to any of its children. In general, using AROWANA tree can reduce a Select2 query into less number of Connect queries than Algorithm 1. For example, consider the query Select2(u_3) in Figure 1. Algorithm 1 would translate it into five Connect queries: Connect(u_3,$B|K|C|W|T$). However, the particular tree in Figure 1 reduces Select2(u_3) into only four Connect queries:

1) Select2(u_3) reduces to two artificial parent Connect queries, Connect(u_3,$BKC|WT$). When the two Connect queries are tested, only Connect(u_3,WT) returns positive, the search can discard the BKC branch and focus only on the WT branch.
2) Connect(u_3,WT) is expanded to two leaf-level Connect queries Connect(u_3,$W|T$). Both queries return positive.
3) Return $\{W, T\}$ as a result for Select2(u_3).

There are cases where the AROWANA tree fails to reduce Select2 efficiently although always correctly. Consider the Select2(u_1) query in Figure 2. The execution of Select2(u_1) follows the same routine as Select2(u_3). But the AROWANA tree reduces Select2(u_1) into seven Connect queries, which is two more than Algorithm 1 in this case.

How to algorithmically organize the AROWANA tree so that the overall average cost of Select2 queries is minimized? Is there a upper bound on the cost of Select2 queries on a given AROWANA tree? Both questions are critical to the validity and feasibility of our proposed idea and we will discuss them in the following sections.

D. Select2 Performance Analysis

This section provides analytical treatment of how the structure of a AROWANA tree can affect typical Select2 query performance. First, we consider a *uniform* AROWANA tree: the children nodes are adopted by their parent nodes in a uniformly random fashion. Assume that the tree has a fertility rate of m: each non-leaf parent has on average m children. The tree has an average height of $K = \log_m(|\Sigma_W|)$. Further suppose that a `userID` u would have connections with n_u out of $|\Sigma_W|$ `brands`.

Proposition 1: Select2(u) has a bounded cost of $K \cdot n_u \cdot m$ in a uniform AROWANA tree with fertility rate m.

Proof: Since $1 \leq n_u \leq |\Sigma_W|$ for $\forall u$, $\exists k \in \mathbb{N}$ and $0 \leq k \leq K$ such that $|\Sigma_{P_k}| \leq n_u \leq |\Sigma_{P_{k+1}}|$. Clearly, k is unique since $\{|\Sigma_{P_j}| \mid 0 \leq j \leq K\}$ is a strictly descending finite sequence of integers. By Dirichlet principle, we can set $C_i(u)$, the upper bound of the number of Connect queries executed to Σ_{P_i} at the i-th level of the AROWANA tree:

$$C_i(u) = \begin{cases} m \cdot n_u, & \text{if } 0 \leq i \leq k \\ m \cdot |\Sigma_{P_i}|, & \text{if } k+1 \leq i \leq K. \end{cases}$$

[2] https://github.com/bitly/dablooms
[3] https://github.com/axiak/pybloomfiltermmap

Then, $C(u)$, the total number of Connect queries that Select2(u), is bounded by $\sum_{i=0}^{K} C_i(u)$. The result follows. ■

$C(u)$ in Proposition 1 is basically a linear bound. It turns out, as shown in Proposition 2, we can greatly improve $C(u)$ to logarithmic if the AROWANA tree is *non-uniform*.

Proposition 2: Suppose that at the i-th level on a non-uniform AROWANA tree for $1 \leq i \leq K$, the child nodes of the same parent p are t_p ("the affinity likelihood") times more likely to have the same membership of any `userID` than any child node whose parent is not p. Further suppose that t_p is proportional to the cardinality in p. Then, the total number of Connect queries that a Select2 query would reduce into, in the non-uniform AROWANA tree, is asymptotically bounded by the logarithmic growth of n_u.

Proof: Exact combinatorial analysis of an upper bound in a non-uniform AROWANA tree would be tedious and probably unnecessary. Instead, we consider a relaxed version where $|\Sigma_{P_i}|$ for $1 \leq i \leq K$ is assumed to be infinite and each parent node at the i-th level has infinite capacity. This assumption relaxes the upper-bound but greatly simplifies the analysis.

The infinite cardinality and infinite capacity assumption immediate translates each level in the non-uniform AROWANA tree into a separate Chinese Restaurant Process (CRP) [15]. CRP models a Chinese restaurant with an infinite number of circular tables, each with infinite capacity. Customer 1 chooses a random table. Customer $n+1$ chooses uniformly at random to sit at one already occupied table, or an unoccupied table. Each round table corresponds to a AROWANA parent node and each customer corresponds to a Connect query. For the i-th level AROWANA tree, $1 \leq i \leq K$, construct a CRP model with parameter α and θ and let $|P_i|$ denote the number of currently occupied tables/parents. The model dictates that customer $n+1$ sits at an unoccupied table/parent with probability $\frac{\theta+|P_i|\alpha}{n+\theta}$, or at an occupied table/parent p with cardinality $|p|$ with probability $\frac{|p|-\alpha}{n+\theta}$. The affinity likelihood t_p can be expressed using α and θ as $\frac{|p|-\alpha}{\theta+|P_i|\alpha}$. Now let X_i be the random variable denoting the number of Connect queries that Select2(u) reduces into. Finding out the $\mathbb{E}[X_i|n_u]$, is then expressed [16] as

$$\mathbb{E}[X_i|n_u] = \frac{\Gamma(\theta+n_u+\alpha)\Gamma(\theta+1)}{\alpha\Gamma(\theta+n_u)\Gamma(\theta+\alpha)} - \frac{\theta}{\alpha},$$

where $\Gamma(\cdot)$ is the Gamma function on real numbers, $0 \leq \alpha \leq 1$, and $\theta > 0$. At $\alpha = 0$, we have

$$\begin{aligned}\mathbb{E}[X_i|n_u, \alpha=0] &= \sum_{k=1}^{n_u} \frac{\theta}{\theta+k-1} = 1 + \theta \sum_{j=1}^{n_u} \frac{1}{\theta+j} \\ &< 1 + \theta \sum_{j=1}^{n_u} \frac{1}{j} = 1 + \theta(\ln n_u + \gamma + o(\frac{1}{2n_u})),\end{aligned}$$

where γ is the Euler-Mascheroni constant. The above inequalities show that $\mathbb{E}[X_i|n_u, \alpha=0]$ is asymptotically bounded by logarithmic growth. ■

E. Exploiting Semantic Affinity

We notice that any AROWANA tree already guarantees asymptotically the temporal lower bound ($O(|\Sigma_W|)$) for Select2 queries (see Table II), even in the worst case where the entire AROWANA tree is searched. Since it is impossible to improve Select2 queries asymptotically, One can seek advantage in the data characteristics of the particular bipartite graph in focus. The plan, as fully explained in the following section, is to exploit the semantic affinity between `userIDs` and `brands` and to maximize the spatial locality of Select2 queries.

To take the proven advantage of non-uniform structure in an AROWANA tree, we are interested in finding some semantic affinity-based hierarchical clusters on the set Σ_W in order to build a non-uniform AROWANA tree. Since each `brand` in Σ_W is a social identity, clustering them is different from clustering numbers or vectors or numbers and requires considerations of inter-`brand` affinity. The Jaccard coefficient matrix is a good option to capture such affinity. Let J be a $|\Sigma_W| \times |\Sigma_W|$ matrix where $J_{p,q} = J(\mathcal{A}_p, \mathcal{A}_q)$, the Jaccard coefficient between set $\mathcal{A}_p$ and set $\mathcal{A}_q$. $\mathcal{A}_q$ denotes the `userIDs` associated with `brand` q. Once matrix J is obtained, a fast community detection algorithm like Clauset-Newman-Moore (CNM) [17] is applied to generate the entire hierarchy of communities/clusters.

However, generating the matrix J exactly would be very expensive because solving for the Jaccard coefficient at each cell involves set operations. To avoid heavy overhead to the overall indexing process, we use reservoir sampling [18] and minHash (the single hash variant) [19] to estimate J. Given a hash function $h(\cdot)$ and a fixed integer k, the *signature* of a set $\mathcal{A}$, $SIG(h(\mathcal{A}))$, is defined as the subset of k elements of $\mathcal{A}$ that have the smallest values after hashing by h, provided that $|c| \geq k$. Then an unbiased estimator of $J(\mathcal{A}_p, \mathcal{A}_q)$ is

$$J\widehat{(\mathcal{A}_p, \mathcal{A}_q)} = \frac{|SIG(h(\mathcal{A}_p \cup \mathcal{A}_q)) \cap SIG(h(\mathcal{A}_p)) \cap SIG(h(\mathcal{A}_q))|}{|SIG(h(\mathcal{A}_p \cup \mathcal{A}_q))|}, \qquad (1)$$

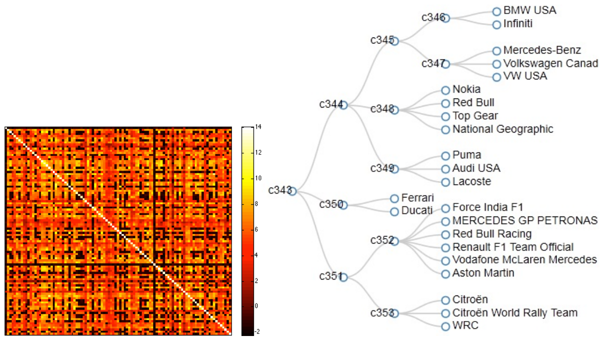
where $SIG(h(\mathcal{A}_p \cup \mathcal{A}_q))$ is the smallest k indices in the union $SIG(h(\mathcal{A}_p)) \cup SIG(h(\mathcal{A}_q))$ and can be resolved in $O(k)$. Figure 2(a) shows how accurately the Jaccard coefficient matrices are estimated. Figure 2(b) shows part of a non-uniform AROWANA tree structure.

Despite claimed scalability and efficiency relative to other even more sophisticated modeling, models in [20] and [21] are very expensive to learn in both temporal and spatial senses. In fact, incorporating one of those modeling techniques for a web-scale heterogeneous bipartite graph would be already more expensive than building a membership B-Tree index on the graph.

F. Deletes and Updates

Deletes and updates to the data have for long troubled high performance indices. Even in proven systems like Sphinx or Lucene, frequent updates and deletes can be easily more expensive than rebuilding the index file entirely [4].

[4]http://lucene.apache.org



(a) (b)

Fig. 2. (a) Jaccard coefficient estimation error matrix (due to minHash estimation) for $k = 50$; (b) Connections in a non-uniform AROWANA. Motorsport subtree shown.

Given such difficulties, AROWANA takes minimalistic implementation of deletes and updates. While it is theoretically possible to have Bloom filters to support removals [14], we simply maintain extra necessary Bloom filters BFM_i for "deleted items" such that whenever an item is marked as removed in BF_i, it is added to BFM_i. Similarly, we can have filters for "re-added", "re-added-then-deleted" items, etc.

V. EXPERIMENTS

A. Baseline Methods

In our experiments, the baseline method **BTree** is implemented in MySQL 5.3 MyISAM engine. A partitioned version, **BTreePar**, is also used in comparisons. Algorithm 2 illustrates the indexing steps of this baseline approach. Algorithm 2 creates two indices: I_W is used for Select1 queries and I_{UW} is used for Select2 and Connect queries. **BTreePar** is similar to **BTree** except that its data file and index file is broken into 1,024 partitions by timestamp and `brand`. Three additional algorithms inspected experimentally are **Arwn**, **ArwnU** and **Lucene**. **Arwn** implements the AROWANA tree and the tree is optimized by inter-brand semantic affinity. **ArwnU** implements the uniform AROWANA tree. **Lucene** indexes the same table t in Algorithm 2 as tokenized text using ElasticSearch [5] with 4 shards of **Lucene** indices. We include **Lucene** in the test because it is an intuitive option to index source file T, which is plain text stream.

B. Datasets

A key motivation in AROWANA is its use of semantic affinity, which is probably best illustrated using social data. In the experiments, Twitter and Facebook graphs collected during 2012 are used. Table III shows the size of the databases we are maintaining using the proposed infrastructure and the amount of data used in the experiments. Figure V-A visualizes the (very skewed) degree distribution in our dataset.

[5]http://www.elasticsearch.org

Algorithm 2: BTree index for bipartite memberships.

Input: T, as in Table 1, t, a row-wise storage
Output: t containing all membership relations
$w \leftarrow$ initial wall node
for *every line* $l \in T$ **do**
  $w \leftarrow$ extract `brand` from line l
  $ls \leftarrow$ extract the list of `userIDs` from line l
  insert $\langle w, u\rangle$ into table t for $\forall u \in ls$
end
build index I_W on $\langle$`brand`$\rangle$ for Select1 queries
build I_{UW} on $\langle$`userID`,`brand`$\rangle$ for Select2 , Connect

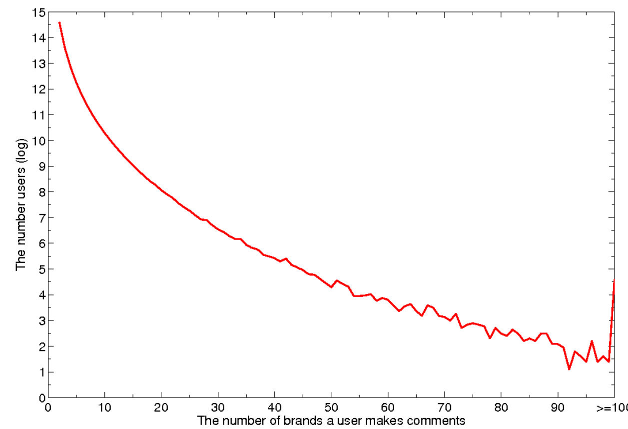


Fig. 3. Facebook user activity distribution (2008 June to 2012 January, vertical axis is in log scale).

TABLE III
DATASETS SUMMARY STATISTICS.

Name	Source	Users	Brands	Connections
T	Twitter	52M	1,270	100M
F1	Facebook	81M	2,898	250M
F2	Facebook	201M	6,553	601M
F3	Facebook	377M	13,740	1.40B

C. Indexing Performance

We compare AROWANA with the baseline methods in the task of performing the three basic kinds of membership queries at different scales. Figure 4 shows the performance comparisons for **Select1**, **Select2**, and **Connect** queries. Among the five compared indexers, there is no clear all-condition winner because each type of query exhibit different temporal characteristics at different data sizes.

Select1 queries are relatively simple to characterize. **BTree**'s retrieving time increases almost linearly as the data size grows and its growth is followed by **Lucene**. The other three indexers' retrieving times only grow marginally even the data size increases by over 10 times. The results, however, do not indicate that the Select1 queries scale better than the theoretical logarithmic bound. A Select1 query is more likely to hit a "small" `brand` with few number of records on larger datasets (F2, F3 in Table III). Only 1,000 random Select1 queries are are tested because an average Select1 query would

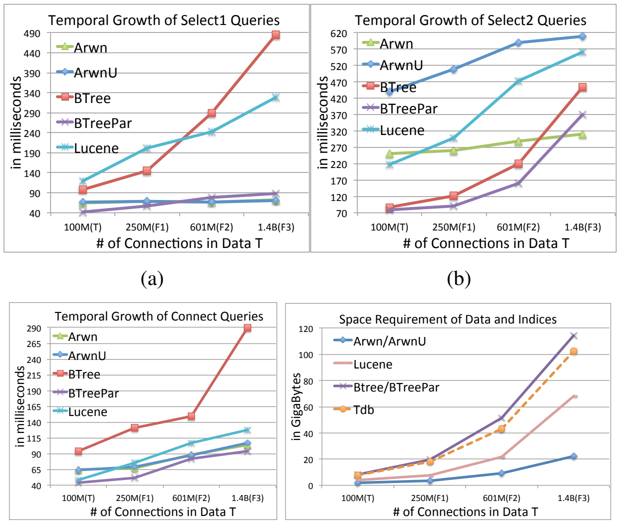


Fig. 4. Comparison of query time on (a) Select1, (b) Select2, and (c) Connect, and (d) Space requirement for indices and data.

retrieve about 1,000 records due to unbalanced graphs.

Select2 performance characteristics shown in Figure 4(b) are quite different from Select1. 1 million random Select2 queries are used. The increasing cost in AROWANA indices is much closer to theoretical prediction (see Table II) than **BTree**, **BTreePar**, and **Lucene**. **Arwn** consistently costs about half of **ArwnU**, which is expected from a semantically optimized AROWANA tree. Eventually, with the largest dataset (F3), **Arwn** outperforms **BTreePar**.

Connect queries perform a lot like Select 1 queries. **BTree**'s retrieving time increases almost linearly as the data size grows. **BTreePar** and **Lucene** grow closer to logarithmically. On the other hand, AROWANA indices (**Arwn** and **ArwnU**) show similar increase in retrieving time. AROWANA indices eventually do not overtake **BTreePar** in query latency, for which a key reason is that Connect can be answered using index files only without touching the data source at all. Once **BTree** index can reside in memory, it is guaranteed to perform well on Connect queries.

D. Practical Impact From Bloom Filters

The use of Bloom filters adds uncertainty to the data structure. Since each leaf node in the AROWANA tree is independently built, it is easy to control the false positive rate for each leaf node independently. Table IV, through F-measure, shows the accuracy of Select2 and Connect queries on AROWANA trees with different Bloom Filter configurations. 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.

E. Index Building Performance Comparison

In addition to query performance, we also measure the spatio-temporal cost to build the index files using different algorithms. Figure 4(d) shows the spatial cost for the AROWANA index, AROWANA uniform index, MySQL B-Tree index, MySQL B-Tree index with partition. It is clear from Figure 4(d) that both AROWANA indices take considerable less space than traditional B-Tree indexing. Figure 4(d) also shows "Tdb", the data size at each stage, which confirms that B-Tree index can grow as fast as (or even faster) than the data being indexed. AROWANA and AROWANA uniform indices are growing sub-linearly, which is in accordance with our theoretical analysis from Table II.

AROWANA index seems to have even greater temporal advantage in the index building. Figure 5 shows the building times for different indexers, number of threads, and data sizes. The unpartitioned B-Tree shows by far the slowest time, which is largely caused by its buffer pool not being big enough for the data/index and is a known issue with systems like MySQL[22]. Like it is demonstrated by [22], the indexing time drops considerably when the data/index is partitioned into 1,024 parts. By partitioning the table on a timestamp key, the database only loads the necessary shard into a buffer (which is much more likely to be able to fit in main memory) at a time and results in a much quicker building time. However,

TABLE IV
F-MEASURES ABOUT QUERIES ON AROWANA TREE.

subgraph	C1	C2	C3	C4	C5	C6
techcrunch	0.548	0.958	0.836	0.893	0.974	0.932
iTunesMusic	0.462	0.962	0.812	0.881	0.983	0.929
google	0.720	0.974	0.931	0.952	0.976	0.964
facebook	0.770	0.974	0.955	0.964	0.975	0.970
intel	0.772	0.980	0.959	0.970	0.981	0.975
netflix	0.772	0.941	0.926	0.934	0.941	0.937
Xbox	0.762	0.972	0.955	0.963	0.972	0.968
eBay	0.948	0.980	0.979	0.980	0.980	0.980
Microsoft	0.941	0.981	0.980	0.980	0.981	0.981
bing	0.915	0.983	0.980	0.982	0.983	0.982
AppStore	0.912	0.974	0.972	0.973	0.974	0.974
iheartradio	0.927	0.980	0.979	0.979	0.980	0.980
amazonmp3	0.969	0.984	0.984	0.984	0.984	0.984
PlayStation	0.950	0.984	0.984	0.984	0.984	0.984
Sprint	0.916	0.967	0.966	0.966	0.967	0.967
XboxSupport	0.731	0.847	0.842	0.844	0.847	0.845
VerizonWireless	0.967	0.978	0.978	0.978	0.978	0.978
nokia	0.970	0.981	0.981	0.981	0.981	0.981

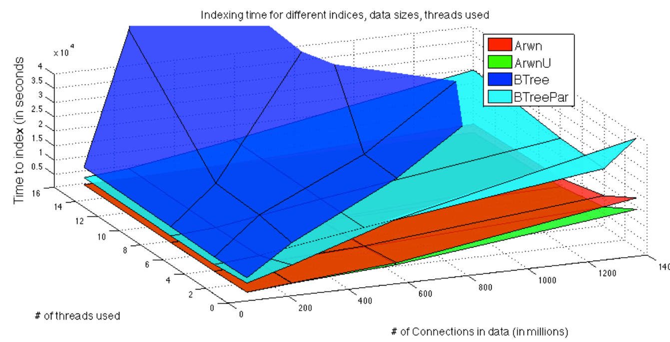


Fig. 5. Index building time for increasingly large data source T and increasingly more threads.

AROWANA and AROWANA uniform indices still take less time to build than the partitioned B-Tree by comfortable margins. AROWANA's main advantage is that its small file sizes, as shown in Figure 4(d), can fit in system's memory. Furthermore, building an index, unlike querying one, involves physically writing the entire index file to disk. In other words, AROWANA can index much faster partly because it writes hundreds of GigaBytes less data to disk than B-Trees in our experiments.

Figure 5 also shows that AROWANA uniform builds marginally faster than AROWANA. The reason is that in order to optimize the AROWANA tree structure by grouping affine brands closer, AROWANA needs to compute inter-`brand` semantic affinities and modify the trees.

VI. ACKNOWLEDGEMENTS

This work is supported in part by the following grants: NSF awards CCF-0833131, CNS-0830927, IIS-0905205, CCF-0938000, CCF-1029166, ACI-1144061, IIS-1343639, and CCF-1409601; DOE awards DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, DESC0005340, and DESC0007456; AFOSR award FA9550-12-1-0458.

VII. CONCLUSION AND FUTURE WORK

We have introduced AROWANA, a data-driven algorithm for indexing large unbalanced bipartite graphs. AROWANA achieves a high-performance efficiency by building an index tree that incorporates the semantic affinity among unbalanced graphs. AROWANA uses probabilistic data structures to minimize space overhead and optimize search. In experiments, we have shown AROWANA's superior scalability and competence in building and retrieving queries over B-Trees and Lucene. In the future, we plan to test AROWANA index on different types of large, dynamic datasets beyond social graphs in order to fully understand its strengths.

REFERENCES

[1] J. Han, "Mining heterogeneous information networks by exploring the power of links," ser. DS '09, 2009, pp. 13–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04747-3_2

[2] Z. Chen, K. Padmanabhan, A. Rocha, Y. Shpanskaya, J. Mihelcic, K. Scott, and N. Samatova, "Spice: discovery of phenotype-determining component interplays," *BMC Systems Biology*, vol. 6, no. 1, p. 40, 2012. [Online]. Available: http://www.biomedcentral.com/1752-0509/6/40

[3] Y. Xie, D. Palsetia, G. Trajcevski, A. Agrawal, and A. Choudhary, "Silverback: Scalable association mining for temporal data in columnar probabilistic databases," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, March 2014, pp. 1072–1083.

[4] Z. Chen, W. Hendrix, H. Guan, I. K. Tetteh, A. Choudhary, F. Semazzi, and N. F. Samatova, "Discovery of extreme events-related communities in contrasting groups of physical system networks," *Data Min. Knowl. Discov.*, vol. 27, no. 2, pp. 225–258, Sep. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10618-012-0289-3

[5] Y. Xie, Z. Chen, A. Agrawal, A. Choudhary, and L. Liu, "Random walk-based graphical sampling in unbalanced heterogeneous bipartite social graphs," in *Proceedings of the 22nd ACM international conference on Conference on information and knowledge management*, ser. CIKM '13. New York, NY, USA: ACM, 2013, pp. 1473–1476. [Online]. Available: http://doi.acm.org/10.1145/2505515.2507822

[6] P. Hebden and A. R. Pearce, "Bloom filters for data aggregation and discovery: a hierarchical clustering approach," in *ISSNIP '05*, 2005.

[7] J. Ledlie, J. M. Taylor, L. Serban, and M. Seltzer, "Self-organization in peer-to-peer systems," ser. EW 10. New York, NY, USA: ACM, 2002, pp. 125–132. [Online]. Available: http://doi.acm.org/10.1145/1133373.1133397

[8] K. Wu, A. Shoshani, and K. Stockinger, "Analyses of multi-level and multi-component compressed bitmap indexes," *ACM TODS*, vol. 35, no. 1, pp. 2:1–2:52, Feb. 2008. [Online]. Available: http://doi.acm.org/10.1145/1670243.1670245

[9] R. Grossi and G. Ottaviano, "The wavelet trie: maintaining an indexed sequence of strings in compressed space," ser. PODS '12, 2012, pp. 203–214. [Online]. Available: http://doi.acm.org/10.1145/2213556.2213586

[10] M.-C. Wu and A. Buchmann, "Encoded bitmap indexing for data warehouses," in *ICDE '98*, 1998, pp. 220–230.

[11] K. Wu, E. Otoo, and A. Shoshani, "On the performance of bitmap indices for high cardinality attributes," ser. VLDB '04. VLDB Endowment, 2004, pp. 24–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=1316689.1316694

[12] R. Pagh and S. R. Satti, "Secondary indexing in one dimension: beyond b-trees and bitmap indexes," ser. PODS '09. New York, NY, USA: ACM, 2009, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/1559795.1559824

[13] H. Maserrat and J. Pei, "Neighbor query friendly compression of social networks," ser. KDD '10. New York, NY, USA: ACM, 2010, pp. 533–542. [Online]. Available: http://doi.acm.org/10.1145/1835804.1835873

[14] P. S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, "Scalable bloom filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255 – 261, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020019006003127

[15] F. L. Wauthier, M. I. Jordan, and N. Jojic, "Nonparametric combinatorial sequence models," ser. RECOMB'11, 2011, pp. 516–530. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987587.1987633

[16] D. M. Blei and P. I. Frazier, "Distance dependent chinese restaurant processes," *J. Mach. Learn. Res.*, vol. 12, pp. 2461–2488, Nov. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1953048.2078184

[17] D. Palsetia, M. Patwary, K. Zhang, K. Lee, C. Moran, Y. Xie, D. Honbo, A. Agrawal, W. Liao, and A. Choudhary, "User-interest based community extraction in social networks," ser. SNAKDD '12, 2012.

[18] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985. [Online]. Available: http://doi.acm.org/10.1145/3147.3165

[19] 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.

[20] C. Wang, J. Han, Y. Jia, J. Tang, D. Zhang, Y. Yu, and J. Guo, "Mining advisor-advisee relationships from research publication networks," ser. KDD '10. New York, NY, USA: ACM, 2010, pp. 203–212. [Online]. Available: http://doi.acm.org/10.1145/1835804.1835833

[21] Y. Sun, J. Han, X. Yan, and P. S. Yu, "Mining knowledge from interconnected data: a heterogeneous information network analysis approach," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 2022–2023, Aug. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2367502.2367566

[22] Scaling mysql writes through partitioning, http://www.slideshare.net/bluesmoon/scaling-mysql-writes-through-partitioning-3397422.