Social Network-Aware Disk Management

Imranul Hoque and Indranil Gupta
Department of Computer Science
University of Illinois at Urbana-Champaign
{ihoque2, indy}@illinois.edu

***Abstract*—Disk access patterns of social networking applications are different from those of traditional applications. However, today's disk layout techniques are not adapted to social networking workloads and thus suffer in performance. In this paper, we first present disk layout techniques that leverage community structure in the social graph to make placement decisions. Second, we build a layout manager called the Bondhu system that incorporates our techniques. We integrate Bondhu into the popular Neo4j graph database engine. Our trace driven experimental results show that the Bondhu system improves the median response time by as much as 48%. While taking the community structure into account yields clear benefits, our results indicate that models with more complexity beyond the social graph may yield low additional benefit.**

I. Introduction

The last few years have seen an unprecedented growth both in variety and in scale of Online Social Networks (OSN). A social network exhibits unique structural properties such as strong community structure and small world phenomena that make disk access patterns of OSN applications different from traditional applications. Our work is motivated by the observation that in order to improve disk access performance of OSN applications at the server side, it is critical to design techniques that take the community structure of OSN into consideration.

There have been several efforts to improve disk performance by careful data organization. The Fast File System improves disk performance by keeping related data blocks and their meta-data together [21]. Multimedia file systems use the organ pipe layout algorithm by tracking the popularity of the objects and keep the hottest object in cylinder zero and place successive cooler records to the left and right respectively [32], [33]. Others track block access patterns and try to place correlated blocks together on the disk [6], [18]. The Free Space File System makes use of the empty space of the disk to replicate blocks according to the observed access patterns [15].

The above approaches are suitable for traditional workloads, such as multimedia file systems, version control systems, and web servers. However, the access patterns in OSNs are quite different from the above access patterns. This is due to many reasons, two of which we briefly discuss here. In a multimedia system popular objects (movies, for example) are popular across all users. On the other hand, in an OSN scenario it is not the case that a few objects dominate globally. Rather, each user accesses her friends' information with a certain probability. Further, existing systems that track the access pattern of blocks and keep related blocks together are less likely to perform well due to the large scale of OSNs. Most of the OSNs consist of millions of users and thus tracking block level access patterns at that scale is not feasible.

This work was supported in part by NSF grant IIS 0841765.

Finding a good disk layout can be helpful in many ways. We mention two specific examples here. Firstly, OSN applications make extensive use of databases at the back-end. Consider a simple table in a database which keeps profile information (name, address, phone, etc.) of users. When a user issues a query to get the name of all of her friends, the disk head has to move to go to the appropriate location in the disk to read her friends' information. A good layout keeps related users' data close by on the disk and hence the disk head movement is reduced. This translates to faster response time in answering the queries. Secondly, consider a custom-built file system for the photo application of a social network which splits the disk into partitions and allocates a partition for a single user. Keeping the partitions organized by a smart layout reduces the disk head movement, since unrelated users rarely access each other's data.

We motivate this further by presenting a visualization of disk block access patterns of a sample OSN application in Figure 1. We use the Facebook New Orleans network graph [30] to build a sample OSN application using the Neo4j graph database [3] (more details are in Section VI). For each user in the social graph, we create a $node$ in Neo4j. Then we store a 400KB data block ($property$ in Neo4j) for each user. Next we write an automated script that logs into the system as a random user and retrieves the data blocks for all of her friends. This is identical to the 'list all friends' action in an OSN. We trace the disk blocks accessed by each request using the $blktrace$ tool [8] and use the $seekwatcher$ tool [20] to visualize the disk block access over time. A dot in Figure 1 depicts a block access at a particular location on the disk at a particular time. We observe from the figure that block accesses are scattered all over the database. This effect is prominent when the queries are issued by users with many friends (around 23 and 46 seconds, for example). Therefore, the disk head has to move a lot to answer this query, which leads to a high response time. Later in Figure 7, we show how social network-aware disk placement performs better for the above workload.

We believe that a social network-aware data organization scheme will improve disk access performance because it changes the random and scattered movement pattern of the

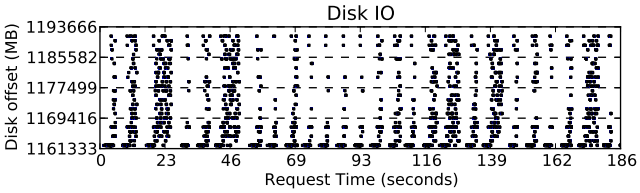


Fig. 1. Blocks accessed in Neo4j when users issue a 'list friend' query

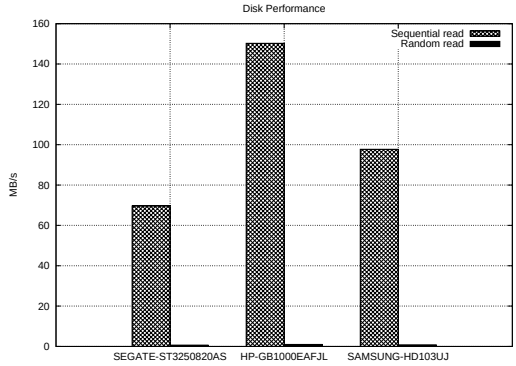


Fig. 2. Sequential vs. random read for 3 disk types

disk head to one which is semi-sequential and confined within smaller regions. To examine how bad the random access performance of a disk is compared to the sequential access performance, we measure disk throughput under both access patterns using the fio benchmarking tool on 3 different hard disks: a 4 year old desktop hard disk (SEGATE), a 2 year old datacenter hard disk (HP), and a new desktop hard disk (SAMSUNG). The results are presented in Figure 2. In all the three disks the random access performance is more than two orders of magnitude worse than the sequential access performance. Therefore, a layout that takes the disk access pattern into account and organizes the data accordingly can improve performance significantly.

Thus, in this paper, we present the design and implementation of the Bondhu[1] System which leverages the social network graph to intelligently layout data on disk. The layout schemes of the Bondhu system improves the disk performance because of three reasons: i) when the user block sizes are small, the data fetched in a single seek contains multiple friends' data, lowering the number of seeks; ii) the disk arm movement is reduced as related data are clustered together – this leads to a lower seek distance (time); iii) rotational latency is improved since the disk has to rotate less to reach the appropriate location for fetching data.

Concretely, we make the following contributions in this paper:

- We present a novel framework for disk layout algorithms based on community detection in a social graph. First, we detect the communities within a social graph. Then, we produce the layout by running a greedy heuristic within and across the communities. To the best of our knowledge, Bondhu is the first system that leverages the social networking graph for efficient data layout in disks.
- We implement our solution into Neo4j, which is a widely used open source graph database. We show through experimentation that the Bondhu system is able to improve response time by as much as 48% when compared to the default layout policy implemented by the file system.
- We also show by experimentation that while taking the social network structure into account helps making better placement decisions, taking the user access patterns into account may not further improve performance much.

The rest of the paper is organized as follows. Section II presents a formal definition of the disk layout problem. Section III discusses the disk layout algorithms which are at the core of the Bondhu system. Section IV gives details of the prototype implementation of the Bondhu system in Neo4j. Section V presents three models for capturing user interactions in OSNs that we use in our experiments. Section VI analyzes experimental results of our prototype implementation. Related works are presented in Section VII. The paper concludes with Section VIII.

II. Problem Definition

Consider N users: $V = \{V_1, V_2, \ldots, V_N\}$, and N consecutive locations on disk denoted by: $L = \{L_1, L_2, \ldots, L_N\}$. Now, consider a function $\delta(V_i, V_j)$ representing the social network.

$$\delta(V_i, V_j) = \begin{cases} 0 & \text{if } V_i, V_j \text{ are not friends} \\ 1 & \text{if } V_i, V_j \text{ are friends} \end{cases}$$

We assume that relationships are symmetric, i.e., $\delta(V_i, V_j) = \delta(V_j, V_i)$ for all (i, j). Define $loc(.)$ to be a one-to-one function which denotes a particular 'layout', i.e., location arrangement, $loc : V \rightarrow L$. There are $N!$ possible $loc(.)$ functions. Further, the $cost$ of a layout from the perspective of a particular user V_i is given by the sum of the difference of the disk locations between the user and all of her friends. The lower the $cost$, the lower the seek distance, and the better the response time. Therefore,

$$cost_i = \sum_{j=1}^{N} [|loc(V_i) - loc(V_j)| * \delta(V_i, V_j)] \quad (1)$$

Therefore, the total cost of a layout is:

$$\begin{aligned} cost &= \frac{\sum_{i=1}^{N} cost_i}{2} \\ &= \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} \{|loc(V_i) - loc(V_j)| * \delta(V_i, V_j)\}}{2} \end{aligned} \quad (2)$$

The lower the cost of a layout, the closer the friends of a user are located on the disk. This speeds up common operations like friend listing, publish-subscribe of wall-posts, etc. Therefore, our goal is to find the layout with the minimum cost.

[1]Bangla word for friend.

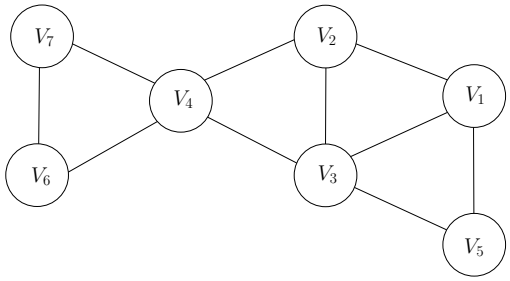


Fig. 3. A sample social graph

TABLE I
COST OF THE LINEAR LAYOUT

Location	L_1	L_2	L_3	L_4	L_5	L_6	L_7
User	V_1	V_2	V_3	V_4	V_5	V_6	V_7
V_1	-	1	2	0	4	0	0
V_2	-	-	1	2	0	0	0
V_3	-	-	-	1	2	0	0
V_4	-	-	-	-	0	2	3
V_5	-	-	-	-	-	0	0
V_6	-	-	-	-	-	-	0
V_7	-	-	-	-	-	-	-

TABLE II
COST OF ONE OF THE OPTIMAL LAYOUTS

Location	L_1	L_2	L_3	L_4	L_5	L_6	L_7
User	V_5	V_1	V_3	V_2	V_4	V_6	V_7
V_5	-	1	2	0	0	0	0
V_1	-	-	1	2	0	0	0
V_3	-	-	-	1	2	0	0
V_2	-	-	-	-	1	0	0
V_4	-	-	-	-	-	1	2
V_6	-	-	-	-	-	-	1
V_7	-	-	-	-	-	-	-

We illustrate the problem with the help of the sample social graph in Figure II with 7 users. Consider the linear layout in Table I: V_1 at L_1, V_2 at L_2, and so on. The users are arranged in the rows and columns according to their layout. An entry (V_i, V_j) in the table is non-zero if there is a link between V_i and V_j in the graph (in other words if V_i and V_j are friends), otherwise it is 0. The non-zero value is the absolute value of the difference of the locations of V_i and V_j (i.e., it is the cost as defined before). Adding up all the values we get the cost of the layout = 18. However, this is not optimal. We present one of the optimal layouts in Table II with cost = 14.

This min-cost social network embedding problem is a variant of the Minimum Linear Arrangement problem, which is known to be NP-hard [13]. The best known heuristic to solve this problem is Simulated Annealing, which itself is computationally infeasible for large graphs [23].

In this paper, we first propose a fast multi-level heuristic to solve this problem, which can handle graphs with millions of nodes. The Bondhu system uses this algorithm to obtain disk layout.

Second, we solve the weighted version of this problem. We use weighted graphs to capture user interactions in the social network. A high edge weight between two users implies that they are more likely to access each other's data and so they should be close by in the disk layout. We use the function $\delta(V_i, V_j)$ to capture the edge weight (w). Thus,

$$\delta(V_i, V_j) = \begin{cases} 0 & \text{if } V_i, V_j \text{ are not friends} \\ w & \text{if } V_i, V_j \text{ are friends} \end{cases}$$

We make one final point about disk geometries before we present our techniques. While disk geometries are often proprietary, manufacturers do present a logical abstraction of the disk which is known as the Logical Block Addressing (LBA) scheme. It is a simple linear addressing scheme where blocks are addressed by an integer index starting from 0. The LBA scheme is essentially a one-dimensional representation of the complex physical geometry of the disk. Disk manufacturers ensure that accessing consecutive blocks in the LBA space is similar to accessing consecutive blocks in the physical geometry. Experimental results [15] also support this claim. Therefore, in this paper we use this simple one-dimensional model of the disk for data layout.

III. DISK LAYOUT ALGORITHM

In this section we present the disk layout algorithm of the Bondhu system. At first we present the intuition behind our proposed algorithm and then explain it in detail in the following subsections.

A. Overview

OSNs are known to exhibit strong community structure. Therefore, we adopt an approach to disk layout algorithms for OSN applications that take the community structure into account. This has multiple benefits. First, the problem space is reduced. So, while making a disk placement decision inside a community we can consider only the members of that community. Second, a bad placement choice will have relatively less impact since the worst possible placement will likely be limited by the community boundary. Third, we can use the existing community detection techniques that are known to find good quality communities in a social graph.

Motivated by these observations, we present the layout algorithm of the Bondhu system. Figure 4 illustrates our approach. The algorithm consists of three modules: i) Community Detection: using existing community detection techniques, we divide the social graph into several communities, ii) Intra-Community Layout: using a greedy heuristic we find a layout for the users within the communities, and iii) Inter-Community Layout: we organize the different communities on the disk by considering inter-community tie strength. These three parts of the framework are discussed below.

B. Community Detection

The goal of the community detection module is to organize the users of the social graph into clusters, so that many edges connect users within the same cluster and relatively few edges connect users in different clusters. The community detection module makes use of existing techniques for graph partitioning and modularity optimization. We select these two algorithm classes because: i) they operate on graphs with large number of vertices, ii) they are known to produce good

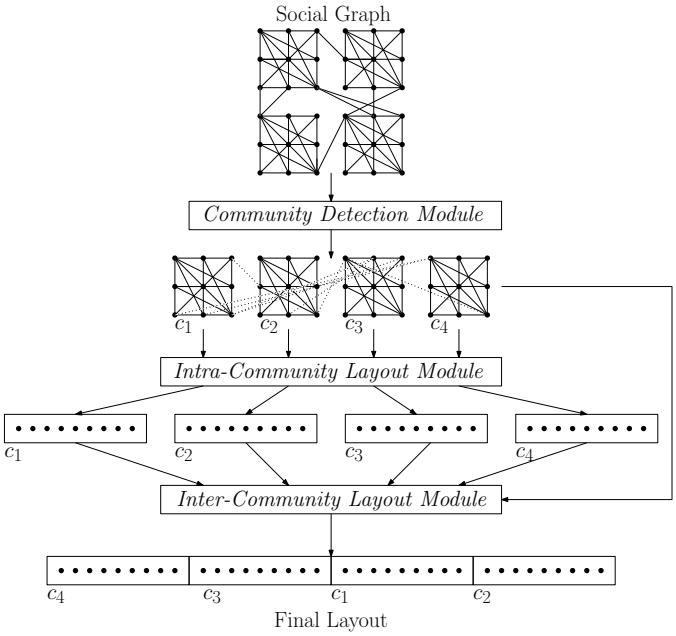


Fig. 4. Overview of the Bondhu system's approach

clusterings, and iii) they are fast, i.e., can find communities on graphs containing millions of nodes within seconds. We briefly discuss the algorithms here.

1) Graph Partition Driven Community Detection: Our graph partition driven community detection algorithm (ParCom) is based on the multilevel k-way hypergraph partitioning scheme of [16], [17]. The goal of ParCom is to partition the social graph into k equal subsets such that the edge-cut is minimized. This is equivalent to minimizing the number of friends in other partitions. ParCom works as follows. First, the social graph is coarsened down to a small number of vertices. In this phase a sequence of smaller graphs is constructed from the original graph by collapsing vertices together using the heavy-edge matching (HEM) technique. The weights of the edges are also recalculated. Then this smaller graph is divided into k-parts using recursive bi-section scheme. Finally the partitions are uncoarsened back to the original graph in steps and at each step the partitions are refined using local refinement heuristics. Fore more details, see [16], [17].

2) Modularity Optimization Driven Community Detection: Our modularity optimization driven community detection algorithm (ModCom) is based on [7]. It is able to detect good quality communities in large networks (118 million nodes).

The modularity of a partition is a scalar value between -1 and 1 that measures the density of intra-community links as compared to inter-community links. More specifically, modularity is defined as the fraction of edges that fall within the communities minus the expected value of the fraction if the edges were randomly distributed (by preserving node degrees). Formally, it is defined as:

$$Q = \frac{1}{2M} \sum_{V_i, V_j} \left[\delta(V_i, V_j) - \frac{k_i k_j}{2M} \right] \sigma(c_i, c_j) \qquad (3)$$

Here, M = number of edges, $\delta(V_i, V_j)$ = weight of the edge between user V_i and V_j, k_i = degree of user V_i (sum of the weights of the links/edges connected to user V_i), c_i = community of user V_i, and $\sigma(c_i, c_j)$ = 1, if $c_i = c_j$, and 0 otherwise.

ModCom works in two phases. In the first phase users are arranged in a random order and each of the users is assigned to a different community. Then for each user V_i the gain in modularity is calculated by removing it from its own community and by assigning it to any of its friends' communities. User V_i is then moved to its friend V_j's community, for which the modularity gain is maximum. In case of no modularity gain, V_i stays in its original community. This first phase is repeated iteratively for all of the users until no further gain in modularity is possible. In the second phase, a new graph consisting of the communities obtained in the first phase is created. Note that the edge weights are recalculated for this phase. After this, the first phase is run again and the process is continued until no further changes are possible. For more details, see [7].

It is important to note that the difference between ModCom and ParCom is that in ModCom the number of communities cannot be controlled explicitly as it can be in ParCom. This affects our later experimentation.

C. Intra-Community Layout

Next, the intra-community layout module takes as input the communities that are produced by the community detection module. For each community it creates a layout for the users within that community. We use a greedy heuristic to find a layout for each of the communities. The heuristic works as follows. We start with the most popular user, i.e., the user with the highest edge degree (=sum of link weights) and place that user in the middle of the disk layout. Next, among all the friends of that user we choose the one that is connected to the user with the heaviest edge. This is to ensure that if two users are strongly connected, they should be placed close by on the layout. In case of a tie, we choose the friend with the higher edge degree (the more popular friend). Intuitively, by adding a popular user early, we provide more choice for the greedy algorithm. We place the friend to the left of the already placed user on the layout. We then create a modified graph by merging the user and her friend. The edges connected to these two users are now connected to the combined node. In case of a common friend of the two users, we assign the weight of the edge between the combined node and the common friend as the sum of the individual edge weights.

Next, among all the friends of this combined node we choose the one that is connected to it with the heaviest edge and place it on the right. We repeat the above steps iteratively by placing the friends to the left and to the right of the already placed users alternatively.

The different components of the algorithm are presented in Algorithm 1 (layout algorithm), Algorithm 2 (finding the maximally connected friend), and Algorithm 3 (creation of the combined node).

Algorithm 1 Calculate Layout L on Graph $G = (V, E(w))$

$enum\{RIGHT = 1, LEFT = 2\}$
$left \leftarrow right \leftarrow \frac{(N+1)}{2}$
$V_c \leftarrow \emptyset$
$direction \leftarrow RIGHT$
//continue until we combine all the nodes
while $size(G) > 1$ **do**
  //find the friend who is maximally connected to V_c
  //in case of $V_c = \emptyset$, return the node with the highest edge degree
  $V_i \leftarrow max\_connected(V_c)$
  //combine V_c and V_i to create a new graph with recalculated edge weights
  $(G, V_c) \leftarrow combine(V_c, V_i, G)$
  //alternate between $left$ and $right$ to place V_i
  if $direction = LEFT$ **then**
    $L_{left} \leftarrow V_i$
    $right \leftarrow right + 1$
    $direction \leftarrow RIGHT$
  else
    $L_{right} \leftarrow V_i$
    $left \leftarrow left - 1$
    $direction \leftarrow LEFT$
  end if
end while

Algorithm 2 $max\_connected(V_c)$

if $V_c = \emptyset$ **then** //initial state
  //return the one with the highest edge degree
  $V_s \leftarrow V_i \,|\, edgeDegree(V_i) \geq edgeDegree(V_j),$
    $\forall V_i \in V, \forall V_j \in V, V_i \neq V_j$
  if $size(V_s) > 1$ **then**
    //return a random one in case of tie
    return $random(V_i) \,|\, V_i \in V_r$
  else
    return V_s
  end if
else //normal case operation
  //select the friend connected to the heaviest edge of V_c
  $V_s \leftarrow V_i \,|\, edgeWeight(V_c, V_i) \geq edgeWeight(V_c, V_j),$
    $\forall V_i \in friend(V_c), \forall V_j \in friend(V_c), V_i \neq V_j$
  if $size(V_s) > 1$ **then** //there is a tie
    //select the one with the highest edge degree
    $V_r \leftarrow V_i \,|\, edgeDegree(V_i) \geq edgeDegree(V_j),$
      $\forall V_i \in V_s, \forall V_j \in V_s, V_i \neq V_j$
    if $size(V_r) > 1$ **then** //there is a tie again
      //return a random one from the list
      return $random(V_i) \,|\, V_i \in V_r$
    else
      return V_r
    end if
  else
    return V_s
  end if
end if

Algorithm 3 $combine(V_c, V_i, G = (V, E(w)))$

//create a new node by joining V_c & V_i
$V'_c \leftarrow createNode(V_c, V_i)$
//add the new node to the set of vertices
$V \leftarrow V \cup V'_c$
//start by deleting the edge between V_c & V_i
$deleteEdge(V_c, V_i)$
for all $F \in friend(V_c)$ **do**
  $w \leftarrow edgeWeight(V_c, F)$
  //delete edges between V_c & its friends
  $deleteEdge(V_c, F)$
  //add edges between the new node & V_c's friends
  $addEdge(V'_c, F, w)$
end for
for all $F \in friend(V_i)$ **do**
  $w \leftarrow edgeWeight(V_i, F)$
  //delete edges between V_i & its friends
  $deleteEdge(V_i, F)$
  //in case of a common friend of V_c & V_i, we already created an edge
  if $isEdge(V'_c, F)$ **then**
    $w' \leftarrow edgeWeight(V'_c, F)$
    //increase the weight of the already created edge
    $setEdgeWeight(V'_c, F, w + w')$
  else //otherwise create a new edge
    $addEdge(V'_c, F, w)$
  end if
end for
$V \leftarrow V - V_c - V_i$ //delete V_c & V_i from the set of vertices
return (G, V'_c)

D. Inter-Community Layout

Our third component is the inter-community layout module. It takes as input: i) the communities produced by the community detection module, and ii) the layout produced within each community by the intra-community layout module. The goal of this component is to create a layout of the communities. This enables us to capture the relationships among different communities. For example, if a community c_i is strongly connected to another community c_j, these two should be placed close by on the disk – this reasoning is similar to the one used for the intra-community layout module.

To create the inter-community layout, we create a graph using the different communities as vertices. The weight of the edge between community c_i and community c_j is calculated as the sum of the weights of the edges from the members of community c_i to the members of community c_j. After creating the community graph we run the same iterative algorithm as the intra-community layout module to find a layout of the communities.

When this is done, we expand the layout within each community, which was previously obtained from the intra-community layout module. This gives us the final disk layout containing all the users of the social graph.

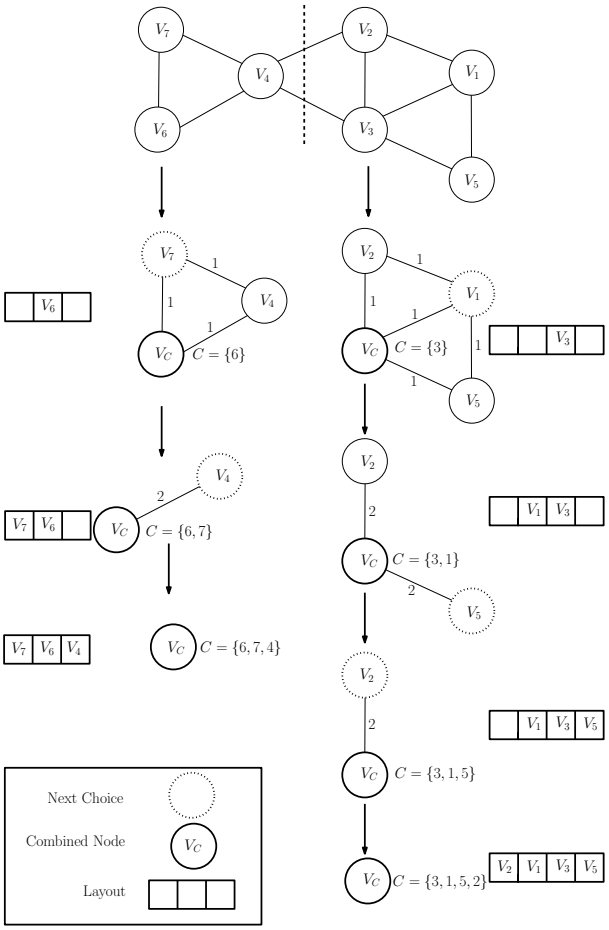
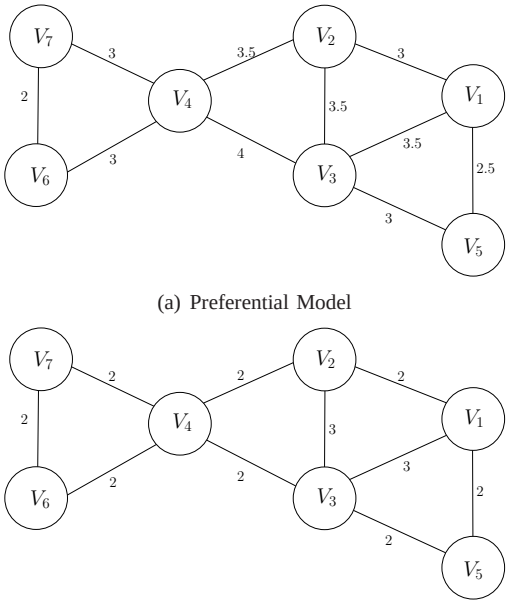


Fig. 5. Working example

Example: We present a working example of our techniques in Figure 5. This is the same graph as shown in Figure II. First, the community detection module splits the graph into two communities: c_1=$\{V_4, V_6, V_7\}$ and c_2=$\{V_1, V_2, V_3, V_5\}$. Then, the intra-community module finds a layout for both of them separately. Let us examine the steps taken by the module for c_2. Here, the first user to be chosen can be either V_3 or V_1, since both of them have the highest edge weight (=3). The algorithm chooses V_3 at random and places it in the middle of the layout. Next, the algorithm considers V_1, V_2, and V_5 (highest edge weight connected to V_3=1). V_1 is chosen since it is the most popular of all (edge degree=3). V_3 and V_1 are combined to $V_{3,1}$ and a new graph is constructed. Now, the algorithm considers V_2 and V_5 (highest edge weight to $V_{3,1}$=2), and V_5 is chosen at random (both V_2 and V_5 are equally popular). $V_{3,1}$ and V_5 are combined to obtain $V_{3,1,5}$, which leaves the algorithm with the last user (V_2) to be placed. The final layout produced for c_2 is: $\{V_2, V_1, V_3, V_5\}$. Likewise, the layout produced for c_1 is: $\{V_7, V_6, V_4\}$. The steps for the inter-community layout module is trivial, since we only have two communities in this example. So, the final layout produced is either $\{c_2, c_1\}$=$\{V_2, V_1, V_3, V_5, V_7, V_6, V_4\}$, or $\{c_1, c_2\}$=$\{V_7, V_6, V_4, V_2, V_1, V_3, V_5\}$ depending on which community is chosen first by the inter-community layout module.



(a) Preferential Model

(b) Overlap Model

Fig. 6. Modeling the social network

IV. Implementation

We implement the Bondhu system as a layout manager for the Neo4j [3] graph database. Neo4j is a very popular and widely used graph database. It is suitable for building OSN applications as it offers a graph-oriented model for data representation. A Neo4j graph consists of nodes, relationships, and properties. Properties are mapping from a string key to a value and can be associated with both nodes and relationships. The part of the Neo4j storage engine that stores properties is known as the PropertyStore.

We modify the PropertyStore of Neo4j so that the records are organized by the layout algorithms of the Bondhu system. Note that we rely on the native file system so our layout decisions are propagated to the disk block level, i.e., the modified PropertyStore database produced by the Bondhu system is stored sequentially on the disk. Therefore, we start with an empty disk and verify with the $davl$ [1] tool that the database file is stored sequentially on the disk at the block level.

Our implementation of the Bondhu system is in Java. The community detection module makes use of the METIS library [2] for the ParCom algorithm.

V. Modeling the Social Network

In this section we present three models to capture user interaction in a social network. We use these models to evaluate our disk layout techniques later in Section VI. These models vary in the way they assign weights to the edges between users.

A. Uniform Model

The uniform model is the simplest of the three models. In this model we assign equal weight (=1) to all social network

edges. In other words, according to this model a user is equally likely to access any of her friends' information. Listing the name of all of the friends of a given user can be viewed as an example of this model.

B. Preferential Model

The preferential model is motivated from the observation that a user with large number of friends is likely to be more active than a user with fewer friends, e.g., make more status updates, post more frequently, etc. In other words, a user with a larger number of friends is more active than a user with fewer number of friends. While browsing, a user is more likely to access the information of the more active friends.

To capture this type of interaction, the weight of the edge (V_i, V_j) should be proportional to the edge degree of V_j. Note that this metric is not symmetric, i.e., if V_j has a higher edge degree than V_i, then the weight of (V_i, V_j) is higher than the weight of (V_j, V_i). On the other hand, disk locality is symmetric in nature and to capture that our social graph models are undirected. Therefore, according to the preferential model the weight of the edge (V_i, V_j) is set to $[edgeDegree(V_i) + edgeDegree(V_j)]/2$. In Figure 6(a) we assign the edge weights according to the preferential model.

C. Overlap Model

The overlap model is motivated by the following observation: two users with a large number of common friends are more likely to share common interests than two users with fewer number of common friends. Therefore, the two users with the larger number of common friends are more likely to access each other's information. In other words, if user V_i has p common friends with V_j and q common friends with V_k, and if $(p > q)$, then V_i is more likely to access V_j's data than V_k's data.

To assign the weight of the edge (V_i, V_j) according to the overlap model, we calculate the number of common friends c between V_i and V_j and set the edge weight as $(c+1)$. We add a 1 to make sure that we do not assign a 0 weight to the edge (V_i, V_j) in case of no common friends, since an edge weight of 0 indicates there is no edge at all. In Figure 6(b), we assign the edge weights according to the overlap model.

VI. Experimental Evaluation

We use the Facebook New Orleans dataset collected in [30]. This dataset contains 63731 users and 817090 links. We assign appropriate weights to the social graph according to our uniform, preferential, and overlap models. Unless otherwise specified the experiments are based on the uniform model.

We run two instances of Neo4j that store the above OSN – one with the integrated Bondhu system handling the data layout and the other one is the unmodified Neo4j. We call the data layout scheme of the unaltered Neo4j the $default$ layout. Based on the method used in the community detection module, the Bondhu system has two data layout schemes built-in: ParCom and ModCom (see Section III for details).

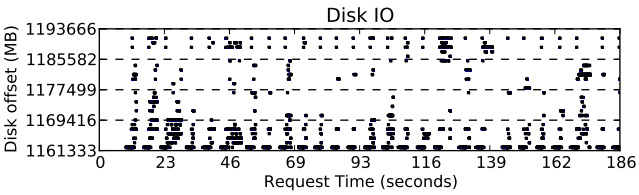


Fig. 7. Blocks accessed in Neo4j with the Bondhu system handling data layout. Compare with Figure 1 (default approach).

We use two metrics to evaluate the data layout schemes of the Bondhu system. The first metric is the cost as defined in Section II (Equation 1). The cost metric measures the spatial clustering of the friends of a user on the disk. Therefore, a lower cost means that the data of related users are placed close by on the disk. Thus, operations like listing friends and wall posts will be faster. Our second metric is response time. This measures the time to fetch data blocks from all the friends of a random user. This captures the performance of an application with our layout schemes. An improvement in the response time metric suggests that the disk is able to handle more requests per unit time and that the user-perceived delay in getting the response to a request is reduced. We next describe the workload we use to measure the response time metric.

Our workload captures the event of listing the friends of a user, which is a very common operation in an OSN. To do this we build a sample OSN application on top of Neo4j. First, we populate the Neo4j database with the social graph. Next, we store a data block ($property$ in Neo4j) for each user in the graph. The Bondhu system handles the data layout automatically beyond that point. Next, we write an automated script that logs into the system as a random user and fetches the data blocks for all the friends of that user. We measure the response time for the whole operation. To make sure that the response time is not adversely affected by other processes accessing the disk at the same time, we carry out the same operation 6 times and take the minimum. We repeat this for 1500 random users. We use the same workload for all of our experiments except for the one on the effect of different models (Section VI-G).

To ensure that the data is served from the disk (and not from the previous cached results in the memory) we flush the memory as follows: First, we use the $sync$ command to write any buffered data to disk. Then, we use the $drop\_caches$ mechanism in the Linux kernel to drop the pagecache, dentries, and inodes from the memory, causing the memory to be free from any cached data. All our experiments are conducted on an HP DL160 compute block with 2 quad core Intel Xeon 2.66 GHz processors, 16 GB of memory, and 2 TB of storage.

We now present our experimental results by tuning different parameters of the Bondhu system.

A. Visualization of Block Access Patterns

To contrast with the disk block access patterns of the default layout presented in Section I, we repeat the same experiment with Bondhu enabled Neo4j system. Here we use ParCom with 64 communities. Per user data size is 400KB as before. We

conduct our workload based measurements and trace the block level I/O for each user request. The visualization of the trace is presented in Figure 7. Each dot shows a read request, its disk offset, and time of request. Here, we observe a significantly better disk block access pattern compared to Figure 1. In Figure 1 the block accesses were scattered, whereas in Figure 7 the block accesses are clustered (prominent at 23, 46, 116–139 seconds). This suggests that the Bondhu system is clustering the related friends' data close by on the disk. This translates to less disk arm movement and thus faster seek and response time.

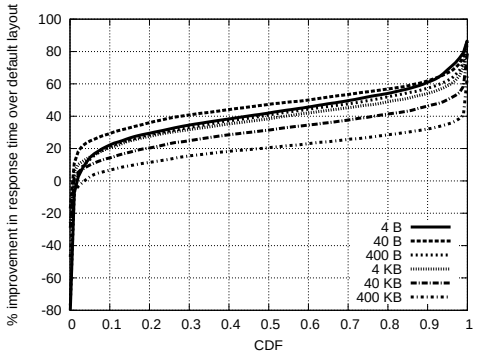
B. Effect of Data Size

In an OSN application the data associated with a particular user can be of different sizes, e.g., it may contain any of name, address, profile picture, wall posts, etc. Therefore, it is important to see the effect of varying user data block sizes on the performance of the layout algorithms. First, we examine the effect of varying data block sizes on the response time metric. Then we present a scatter plot to show the correlation between the improvement in the cost metric and the improvement in the response time metric. This shows to what extent the improvement in data locality translates to the improvement in response time.

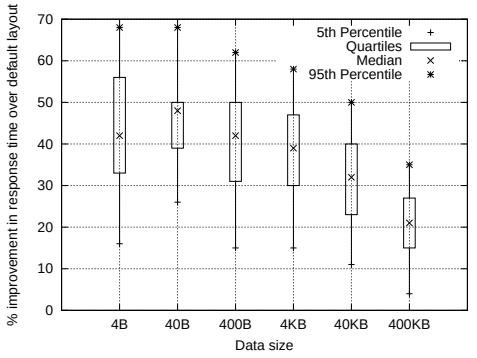
For this experiment we create data blocks of 4B, 40B, 400B, 4KB, 40KB, and 400KB for each of the users and conduct our workload based measurement. We use ParCom with 64 communities, compare it with the default layout, and plot the improvement in the response time metric (the lower the response time compared to the default layout, the more is the improvement). We plot the CDF of the improvement for the different data sizes in Figure 8(a) and the 5th percentile, quartiles, and the 95th percentile of the same results in Figure 8(b).

We see a 22% to 48% median improvement in response time compared to the default layout across various data sizes. The Bondhu layout manager performs best when the user data size is 40B. When the data sizes increase from 4B to 40B we see an increase from 42% to 48% in the median improvement compared to the default layout. Beyond that, the improvement percentage decreases and at 400KB the median improvement reaches 22%.

The reasoning for the above behavior is as follows. The native file system reads data in chunks of 4KB blocks. Therefore, when user block sizes are small, a file system read fetches multiple users' data together. For example, with 4B user block size, a read yields around 1024 users' data. With 40B user block size, a read yields around 102 users' data, and with 400B user block size, a read yields around 10 users' data. Due to the randomness of the data placement scheme of the default layout, the expected number of friends present per read decreases by a factor of 10 when data block sizes grow from 4B to 40B to 400B. For Bondhu, however, the expected number of friends present per read does not decrease much when data block sizes grow from 4B to 40B. This is due to the clustering property of Bondhu. In contrast, when the block



(a) CDF



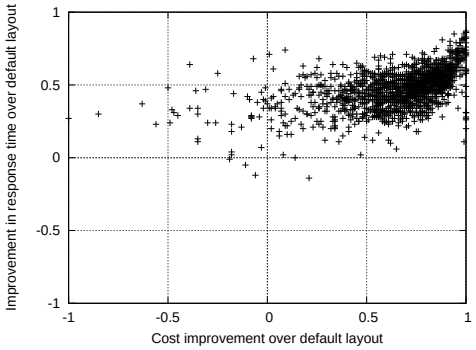
(b) 5th percentile, quartiles, and 95th percentile

Fig. 8. Percentage of improvement in response time compared to the default layout for various data sizes (without caching)

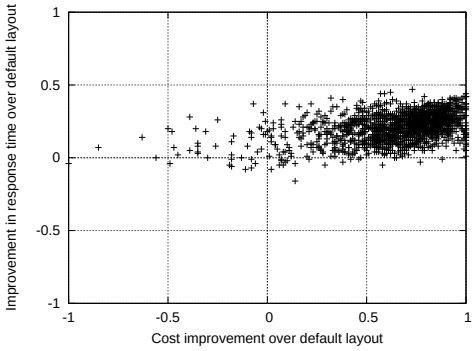
sizes grow from 40B to 400B, the expected number of friends present in per read decreases dramatically. Therefore, we see the drop in improvement after 40B.

In summary, when user data size is smaller than the file system block size, the improvement is high due to fact that a single file system read yields more correlated data. So, the number of seeks required to fetch all friends' data is reduced. This phenomenon begins to vanish when user data sizes grow larger than the file system block size. Beyond that point, the Bondhu system improves performance by reducing the seek distance.

Next, Figure 9 examines the correlation between the improvement in the cost metric and the improvement in the response time. This shows how the smart placement decision of the Bondhu system translates to better application-level performance. As defined in Section II, the cost metric for a user is the sum of differences between the user and her friends' data location. We calculate the cost metric for the users using the placement in both the default layout and the ParCom layout. A larger cost denotes that the friends of a user are far away in the disk. We then calculate the fraction of improvement by using the ParCom layout scheme of the Bondhu system over the default layout. For the corresponding users we measure the fraction of improvement in response time metric and plot the results using a scatter plot. The results are presented in Figures 9(a) and 9(b) for two different data sizes. Figure 9(a) shows good correlation since most points are along the $x = y$ line. For Figure 9(b) the correlation is

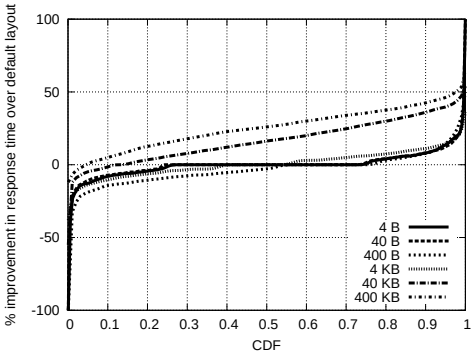


(a) Data size: 40B

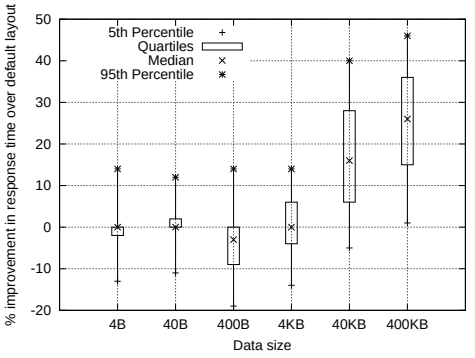


(b) Data size: 400KB

Fig. 9. Correlation between cost improvement and response time improvement (without caching)



(a) CDF



(b) 5th percentile, quartiles, and 95th percentile

Fig. 10. Percentage of improvement in response time compared to the default layout for various data sizes (with caching enabled)

less prominent because of the prior discussion.

C. Effect of Caching

In the previous section we ensure that all the requests are served from the disk and not from the memory. However, serving results from the memory reduces the response time by a large fraction for any application. So, we enable caching for both Neo4j and Bondhu. The results presented in this section examine the effect of caching on the response time metric.

We use the same workload as discussed earlier, but without flushing the cache between successive user requests. A user issues 10 successive requests to fetch the data blocks of all of her friends. As before we conduct this experiment for 1500 randomly selected users.

As with the previous experiment, we plot the CDF of the improvement in response time for the different data block sizes in Figure 10(a) and the 5th percentile, quartiles, and the 95th percentile of the same results in Figure 10(b). When the data size is small we do not see much improvement using our layout scheme. As the data sizes increase from 4KB to 40KB to 400KB the benefit of using the Bondhu system kicks in as seen by the rise in median response time improvement from 0% to 16% to 26% respectively. This is because when the data sizes are small, the information of all the users can be kept in memory. Therefore, requests for data can be readily served from the memory for the default layout as well as for the Bondhu layout schemes. When the data size grows beyond some threshold (40KB here), all the user data blocks cannot

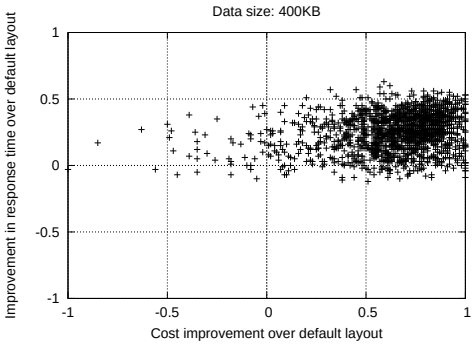
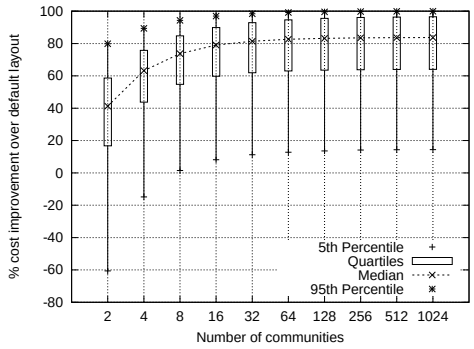


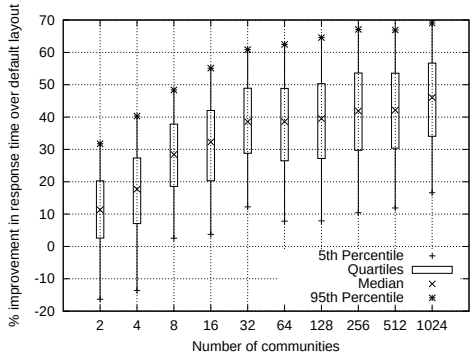
Fig. 11. Correlation between cost improvement and response time improvement (with caching)

be kept in memory. If the data cannot be served from memory, it has to be fetched from disk and thus the previous section's described behavior kicks in.

To investigate whether the improvement in response time for larger data sizes is indeed due to the placement decisions by the Bondhu system, we present a scatter plot of the improvement in response time vs. the improvement in the cost metric in Figure 11. This is similar to the one presented in the previous section but with caching enabled. We observe a fair amount of correlation between the improvement in the two metrics in this case as well. However, the correlation is not as strong as in Figure 9(b). With caching enabled, a fraction of the friends' data will be readily available in the memory. For the already cached data no disk read will be performed.



(a) Percentage of cost improvement over default layout



(b) Percentage of improvement in response time over default layout

Fig. 12. Performance of ParCom

In summary, the worst case median improvement achieved by Bondhu is 0% (small data sizes with caching) and the best case improvement in 48% (medium data sizes without caching). Thus, it is always preferable to use Bondhu.

D. Effect of Number of Communities in ParCom

One of the parameters that can be tuned in ParCom is the number of communities. The fewer the number of communities, the larger the size of a community. For instance, with 1 community, the layout decision is solely handled by the intra-community layout module. With an increase in the number of communities, the inter-community layout module influences layout more. For a given social network graph, we desire to tune the number of communities in such a way that the best disk layout is obtained.

We vary the number of communities in ParCom and examine the improvements in the cost metric and in the response time metric over the default layout. The workload is the same as discussed earlier and the data size per user is 4KB. The results are presented in Figures 12(a) and 12(b) respectively. In Figure 12(a) we see that as the number of communities increases from 2 to 32 we experience a steady improvement in the cost metric. When we have fewer communities, the intra-community detection module is mostly responsible for the layout and the Bondhu system does not capture the community structure of the social graph. Therefore, the improvement grows quickly as the number of communities is increased. However, this curve hits a knee at 64 communities and plateaus out thereafter. This is because the community detection module has lower marginal utility in finding more community structure in the graph towards the right end of the plot.

A similar pattern is observed in Figure 12(b), where we plot the improvement in the response time metric over the default layout for different number of communities. When the number of communities is 2, the median improvement in the response time metric is around 11% for ParCom and this grows quickly. The knee is reached at 32 communities, where the response time of ParCom is 40% lower than that of the default layout. The reasoning is the same as in the previous paragraph.

E. Performance of ModCom

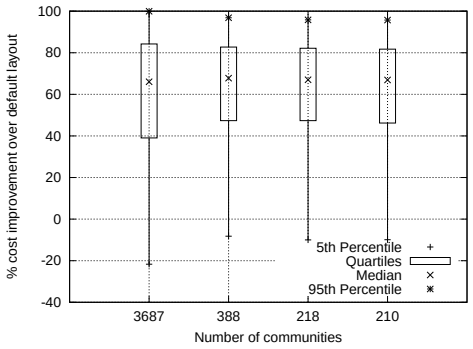
We now focus on ModCom and examine the improvements in the cost metric and the response time metric over the default layout. Unlike ParCom we cannot set the number of communities in ModCom since the number of communities evolve depending on the structure of the social graph. However, ModCom produces communities at different granularities. Recall that the algorithm is iterative – at level 0, there are as many communities as the number of nodes. Level $(i+1)$ combines the communities of level i, and produces fewer communities. We configure the Bondhu system so that it can organize the disk layout based on the communities found at any level. For example, if we set level=2, then the community detection module produces 388 communities which is then fed to the intra- and intra-community layout modules in succession. The workload and the metrics considered are same as the other experiments. Data block size for each user is set to 4 KB.

In Figure 13(a) we present the improvement in cost metric compared to the default layout for varying number of communities found by the community detection module. We observe that unlike ParCom, the median improvement ($\approx 67\%$) does not change much by varying the number of communities. This is because ModCom does not produce a community until it has found a good one (based on the value of the modularity). For the same reason, a flat pattern is observed for the response time metric in Figure 13(b).

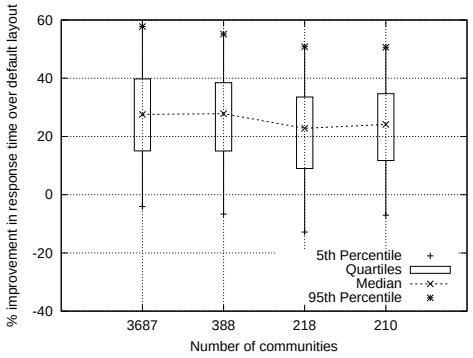
F. Organ Pipe Layout

Next, we compare our layout algorithm with the popular organ pipe algorithm [32], [33], which is used in multimedia file systems. Given a set of records $R_1, R_2, \ldots, R_N$ with global access probabilities $P_1, P_2, \ldots, P_N$, and $\sum_{i=1}^{N} P_i = 1$, the organ pipe algorithm places the record R_i with the largest P_i in the middle and then iteratively places the record with the next largest P_i alternatively to the left and to the right of the already placed record(s). So, according to the organ pipe scheme, the most popular user (the user with the largest edge degree) is placed in the middle and the users with the next largest edge degrees are placed alternately to the left and to the right of the already placed user(s).

We modify the Bondhu system to organize data according to the organ pipe scheme and compare it with ParCom (number of communities=64). Figure 14 plots the CDF of the improvements of the response time metric compared to the default



(a) Percentage of cost improvement over default layout



(b) Percentage of improvement in response time over default layout

Fig. 13. Performance of ModCom

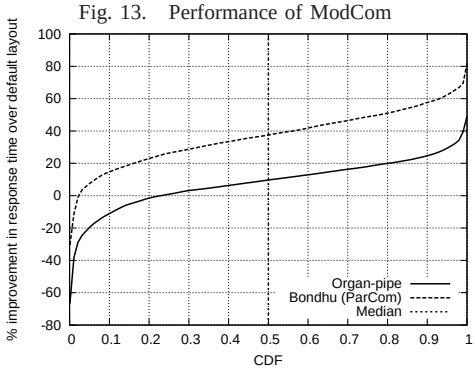


Fig. 14. Comparison with organ pipe layout

layout for both. The data block size for each user is $4KB$ and the workload is the same as the preceding experiments.

The organ pipe is better than the default layout by 10% (on average), but ParCom outperforms the default layout by 38%. The organ pipe scheme assumes that popular users are popular across the system, which is not valid for an OSN. An OSN has a very specific community structure and in this structure popular users are popular only among their friends.

G. Effect of Different Models

So far all the experimental results are based on the uniform model. In this section we present results using the different models presented in Section V: i) the preferential model, ii) the overlap model, and iii) the uniform model. To provide as a baseline for comparison we also present results using the default layout. We use the same social graph as the previous experiments. Data block size for each user is 4KB. The Bondhu system takes the model as the input, creates a layout using that model, and organizes the data according to that layout.

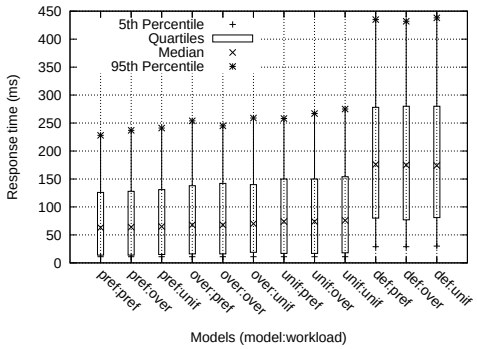


Fig. 15. Effect of different models

We use 3 different workloads based on the graph models. First, in the uniform workload we randomly select a user who issues a request to access one of her friends' blocks at random. Second, in the preferential workload the randomly selected user issues a request to access a friends' data blocks with probability proportional to the friend's degree. Third, in the overlap workload the randomly selected user issues a request to access one of her friends' data blocks with probability proportional to the number of common friends with the friend. In each of these workloads a user issues 1000 successive requests and the response time is measured. Each measurement is taken 10 times and we take the minimum response time. We conduct this experiment for 1000 users in total.

We run each of the 3 workloads on the 4 different layouts and present the results in Figure 15. Each run of the experiment is denoted by $(model : workload)$, where model denotes the models we use: {preferential, overlap, uniform, default} and workload denotes the workloads we use: {preferential, overlap, uniform}. We plot the 5th percentile, quartiles, and the 95th percentile of the response time for all of the runs.

We make three observations from this plot. First, the default layout performs twice worse than any of the other models (median response time: 175ms). Second, the performance of the layout produced by the uniform model is quite comparable to the performance of the layout produced by the preferential and overlap models. Third, the performance of a specific layout does not vary much over the different workloads.

One directional conclusion from these observations is that although it is possible to create complex models (e.g., [5]) to capture user interactions in a social network, often the simplest model (such as the uniform model) is sufficient to make important disk layout decisions. Taking more complex models into account may yield little added benefit for the amount of effort involved. The social graph structure is the biggest influence on disk performance.

VII. Related Works

Data organization techniques for improving disk performance broadly fall into two categories: i) access pattern-

oblivious, and ii) access pattern-aware. Access pattern-oblivious techniques include placing data and meta-data together as in the Fast File System and its variants [21], [9], [29], writing data sequentially to contiguous disk segments as in the Log-structured File System [27], and explicitly grouping small files together on disk as in C-FFS [12]. Access pattern-aware techniques can be further categorized into three types depending on the level of abstraction they work at: i) cylinder level [19], [31], [28], ii) block level [4], [18], [15], [6], [14], and iii) file system level [22].

The position of Bondhu is in the middle of these two extremes. On one hand, it is not access pattern-oblivious in the sense that it captures the community structure of the social network. On the other hand, it is not completely access pattern-aware in the sense that it does not make placement decisions based on the real traffic between users.

Aside from data organization, disk performance can be significantly improved using intelligent prefetching and caching techniques [18], [10]. C-Miner [18], for example, uses data mining techniques to learn the block access patterns and uses that information to make prefetching and cache replacement decisions. The Bondhu system can be extended to make social network-aware prefetching decisions, which remains as one of our future works.

With the recent growth of OSNs, many focused on partitioning the social graph to make OSN applications scalable [24], [26], [25]. SPAR [25], for example, uses partitioning and replication techniques to reduce network traffic across servers. Bondhu, on the other hand uses partitioning and community detection techniques for disk performance improvement. An excellent survey on existing community detection techniques is available at [11].

VIII. Conclusion

In this paper, we presented techniques for disk layout organization for online social networking applications by taking the community structure of the social graph into account. We incorporated our techniques into the Neo4j graph database by building a layout manager called the Bondhu system. Experimental results with realistic workloads exhibited significant improvement in cost and response time compared to the default layout. Our results also indicate that models with more complexity beyond the social graph may yield low additional benefit.

References

[1] "Disk Allocation Viewer for Linux," http://davl.sourceforge.net/.

[2] "METIS: Family of Multilevel Partitioning Algorithms," http://glaros.dtc.umn.edu/gkhome/views/metis.

[3] "Neo4j: The Graph Database," http://neo4j.org/.

[4] S. Akyürek and K. Salem, "Adaptive Block Rearrangement," *ACM Transactions on Computer Systems*, vol. 13, pp. 89–121, May 1995.

[5] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, "Characterizing User Behavior in Online Social Networks," in *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference*, ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 49–62.

[6] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-optimizing Storage Systems," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, 2009, pp. 183–196.

[7] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast Unfolding of Communities in Large Networks," *Journal of Statistical Mechanics: Theory and Experiment*, p. P10008, 2008.

[8] A. D. Brunelle, *blktrace User Guide*, February 2007.

[9] R. Card, , T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[10] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," in *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007, pp. 20:1–20:14.

[11] S. Fortunato, "Community Detection in Graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[12] G. Ganger and M. F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," in *Proceedings of the 1997 USENIX Technical Conference*, 1997, pp. 1–17.

[13] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some Simplified NP-complete Problems," in *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 1974, pp. 47–63.

[14] W. W. Hsu, A. J. Smith, and H. C. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Transactions on Computer Systems*, vol. 23, pp. 424–473, November 2005.

[15] H. Huang, A. Hung, and K. G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption," in *Proceedings of 20th ACM Symposium on Operating System Principles*. ACM Press, 2005, pp. 263–276.

[16] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.

[17] ——, "Multilevel k-way Partitioning Scheme for Irregular Graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[18] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," in *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, 2004, pp. 173–186.

[19] Y. Manolopoulos and J. G. Kollias, "Optimal Data Placement in Two-Headed Disk Systems," *BIT*, vol. 30, no. 2, pp. 216–219, 1990.

[20] C. Mason, "Seekwatcher," http://oss.oracle.com/~mason/seekwatcher/.

[21] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, pp. 181–197, 1984.

[22] J. A. Nugent, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "Controlling your PLACE in the File System With Gray-Box Techniques," in *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, 2003, pp. 311–324.

[23] J. Petit, "Experiments on the Minimum Linear Arrangement Problem," *Journal of Experimental Algorithmics*, vol. 8, December 2003.

[24] J. M. Pujol, V. Erramilli, and P. Rodriguez, "Divide and Conquer: Partitioning Online Social Networks," *CoRR*, vol. abs/0905.4918, 2009.

[25] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The Little Engine(s) that Could: Scaling Online Social Networks," in *Proceedings of the ACM SIGCOMM 2010*, 2010, pp. 375–386.

[26] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez, "Scaling Online Social Networks without Pains," in *Proceeding of the 5th International Workshop on Networking Meets Databases*, October 2009.

[27] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, pp. 26–52, February 1992.

[28] C. Ruemmler and J. Wilkes, "Disk Shuffling," HP Technical Report, HPL-91-156, Tech. Rep., 1991.

[29] S. C. Tweedie, "Journaling the Linux ext2fs Filesystem," in *LinuxExpo*, 1998.

[30] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the Evolution of User Interaction in Facebook," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks*, August 2009.

[31] P. Vongsathorn and S. D. Carson, "A System for Adaptive Disk Rearrangement," *Software: Practice and Experience*, vol. 20, pp. 225–242, March 1990.

[32] C. K. Wong, "Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems," *ACM Computing Surveys*, vol. 12, pp. 167–178, June 1980.

[33] ——, *Algorithmic Studies in Mass Storage Systems*. New York, NY, USA: W. H. Freeman & Co., 1983.