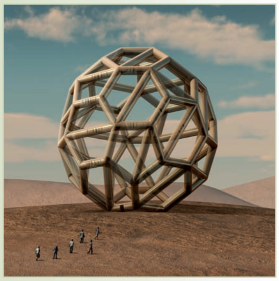

Disk Layout Techniques for Online Social Network Data

Social networking applications' disk access patterns differ from those of traditional applications. However, today's disk layout techniques aren't adapted to social networking workloads, and thus their performance suffers. The authors' disk layout techniques leverage community structure in a social graph to make placement decisions that optimize read latency. Their layout manager, Bondhu, incorporates these techniques and is integrated into the popular Neo4j graph database engine. Experimental results show that Bondhu improves the median response time for online social network operations by as much as 48 percent.

Imranul Hoque and Indranil Gupta
University of Illinois, Urbana-Champaign

The last few years have seen unprecedented growth in the variety and scale of online social networks (OSNs). Disk performance is critical for these systems because improved performance translates to high throughput and low latency for OSN operations. These networks' unique properties — such as strong community structure and small-world phenomena — make the disk access patterns of OSN applications different from those of traditional applications. In order to improve such applications' disk access performance at the server side, we need techniques that take OSNs' community structure into consideration.

Several efforts have aimed to improve disk performance via careful data organization (see the "Related Work in Data Organization Techniques" sidebar). However, although these approaches are suitable for traditional workloads, such as multimedia file systems, version control systems, and Web servers, they are less likely to perform well for OSN workloads. For one thing, in a multimedia system, popular objects (such as movies) are popular across all users. In contrast, in an OSN scenario, rather than a few objects dominating globally, each user accesses his or her friends' information with a certain probability. Furthermore, most OSNs have millions of users, and existing systems that track block access patterns and keep related blocks together aren't likely to perform well at this scale.

Bondhu (the Bangla word for "friend") provides a framework for disk layout algorithms based on community detection in a social graph. Bondhu's layout schemes lower the number of seek operations for small user block sizes by fetching multiple friends' data

Related Work in Data Organization Techniques

Data organization techniques for improving disk performance broadly fall into two categories: access-pattern-oblivious and access-pattern-aware. Access-pattern-oblivious techniques include placing data and metadata together,[1] writing data sequentially to contiguous disk segments,[2] and explicitly grouping small files together on disk.[3] We can categorize access-pattern-aware techniques into three types depending on the level of abstraction at which they work: cylinder-level,[4] block-level,[5,6] and file-system-level.[7]

Bondhu falls in the middle of these two extremes. It isn't access-pattern-oblivious in the sense that it captures an online social network's community structure. However, it isn't completely access-pattern-aware because it doesn't make placement decisions based on the real traffic between users.

Intelligent prefetching and caching techniques can also significantly improve disk performance.[5,8] C-Miner, for example, employs data mining techniques to learn block access patterns and uses that information to make prefetching and cache replacement decisions.[5] We can extend the Bondhu system to make social-network-aware prefetching decisions, which remains part of our future work.

References

1. M.K. Mckusick et al., "A Fast File System for UNIX," *ACM Trans. Computer Systems*, vol. 2, no. 3, 1984, pp. 181–197.
2. M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems*, vol. 10, no. 1, 1992, pp. 26–52.
3. G. Ganger and M.F. Kaashoek, "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," *Proc. 1997 Ann. Usenix Technical Conf.* (ACT 97), Usenix Assoc., 1997, pp. 1–17.
4. C. Ruemmler and J. Wilkes, *Disk Shuffling*, tech. report HPL-91-156, Hewlett-Packard Labs, 1991.
5. Z. Li et al., "C-Miner: Mining Block Correlations in Storage Systems," *Proc. 3rd Usenix Conf. File and Storage Technologies* (FAST 04), Usenix Assoc., 2004, pp. 173–186.
6. M. Bhadkamkar et al., "BORG: Block-reORGanization for Self-Optimizing Storage Systems," *Proc. 7th Usenix Conf. File and Storage Technologies* (FAST 09), Usenix Assoc., 2009, pp. 183–196.
7. J.A. Nugent, A.C. Arpaci-dusseau, and R.H. Arpaci-dusseau, "Controlling Your PLACE in the File System with Gray-Box Techniques," *Proc. 2003 Ann. Usenix Technical Conf.* (ATC 03), Usenix Assoc., 2003, pp. 311–324.
8. X. Ding et al., "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," *Proc. 2007 Ann. Usenix Technical Conf.* (ATC 07), Usenix Assoc., 2007, pp. 20:1–20:14.

in a single seek. Bondhu also reduces disk arm movement by clustering related data together, leading to a lower seek distance. Finally, Bondhu improves rotational latency because the disk has to rotate less to reach the appropriate location for fetching data.

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 and integrate our solution in Neo4j (http://neo4j.org), a widely used open source graph database.

Optimizing the Disk Layout

Finding a good disk layout can help optimize read latency in OSNs. Let's look at two specific examples. First, consider a simple database table that stores users' profile information (name, address, phone number, and so on). When users issue a query to get the names of all their friends, the disk head must seek the appropriate location in the disk to read those friends' information. In the absence of a layout manager, related users' data will be scattered all over the disk, thus increasing disk-head movement. On the other hand, a good layout might reduce this movement by keeping related users' data near each other on the disk, thus lowering the response time. Second, consider a custom-built file system for a social network's photo application that splits the disk into partitions and allocates one partition for each user. Keeping the partitions organized with a smart layout reduces disk-head movement because unrelated users rarely access each other's data.

A social-network-aware data organization scheme can improve disk access performance because it changes the disk head's random and scattered movement pattern to one that's semi-sequential and confined within smaller regions. A benchmarking experiment we conducted using the *fio* tool on three different hard disks showed that random disk access performance was more than two orders of magnitude worse than sequential access performance. So, a layout that accounts for the disk access pattern and organizes the data accordingly could improve performance significantly.

With a large OSN, a social graph must be partitioned across several machines, a problem addressed elsewhere.[1] Here, we focus on how the data layout should be organized on a single disk (that is, our approach is complementary and orthogonal to this prior work).

Problem Definition

Consider N users, $V = \{V_1, V_2, \ldots, V_N\}$, and N consecutive locations on a disk, denoted by

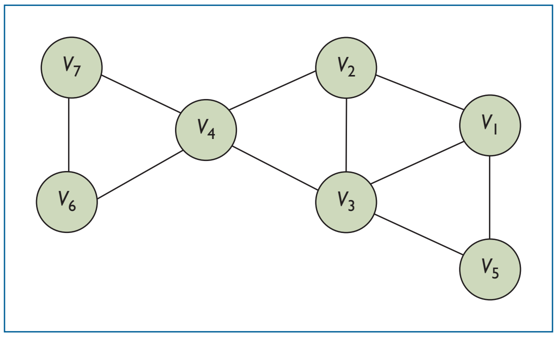

Figure 1. A sample social graph. We use this graph to illustrate the min-cost social network embedding problem.

$L = \{L_1, L_2, \ldots, L_N\}$. 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 — that is, $\delta(V_i, V_j) = \delta(V_j, V_i)$ for all (i, j). This is the case in most popular OSNs such as Facebook and Orkut. We define $loc(.)$ to be a one-to-one function that denotes a particular "layout" — that is, loc: $V \rightarrow L$. There are $N!$ possible $loc(.)$ functions. Furthermore, we give a layout's cost from the perspective of a particular user V_i as the sum of the difference of the disk locations between the user and all of his or her friends. So,

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

A layout's total cost is thus

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

Our goal is to find the layout with the minimum cost; the lower a layout's cost, the closer a user's friends are located on the disk. This speeds up common OSN operations such as friend listing, publish–subscribe for wall posts, and so on by lowering seek time and thus response time for these operations.

We illustrate the problem with the help of the sample social graph in Figure 1, which consists of seven users. Consider the linear layout in Table 1: V_1 at L_1, V_2 at L_2, and so on. Users are arranged in rows and columns according to their layout. An entry (V_i, V_j) in the table is nonzero if a link exists between V_i and V_j in the graph (in other words, if V_i and V_j are friends); otherwise, it's 0. The nonzero value is the absolute value of the distance between the locations of V_i and V_j (that is, it's the cost as defined in Equation 1 for a specific j). Adding up all the values, we get a layout cost of 18. However, this isn't optimal. Table 2 presents an optimal layout with a cost of 14.

This *min-cost social network embedding problem* is a variant of the minimum linear arrangement problem, which is known to be NP-hard.[2] The best-known heuristic for solving this problem is simulated annealing, which is computationally infeasible for large graphs.[3]

Thus, we propose a fast multilevel heuristic algorithm to solve this problem, which can handle graphs with millions of nodes. Bondhu uses this algorithm to obtain disk layout and places the user blocks accordingly.

Our problem formulation generalizes to handle weighted graphs, which we use to capture user interactions in the OSN. A high edge weight between two users implies that they're more likely to access each other's data and should preferably be near each other in the disk layout. We use the function $\delta(V_i, V_j)$ to capture the edge weight (w). Thus,

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

Operations in OSNs span a spectrum, from those involving all of a given user's friends (for example, list all friends) to those involving a single friend (such as viewing one friend's wall post). In between are operations involving some subset of the user's friends (such as status updates from friends). Because exploring all the points on this spectrum is infeasible, we focus only on the two extremes. The performance of operations in between the extremes can be interpolated. Thus, our primary focus is on the "list all friends" operation and individual friend accesses.

We classify OSN user data as *frequently updated data*, such as wall/timeline posts and status updates, or *mostly static data*, such as profile information, profile pictures, and friend lists. Bondhu focuses primarily on the second

Table 1. Cost of the linear layout for Figure 1.

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 2. Cost of an optimal layout for Figure 1.

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	—	—	—	—	—	—	—

category, which also includes information about where the first type of data resides — for example, in addition to basic user profile information, a Bondhu block holds pointers to the hostnames/disk addresses of all photos for that user. This implies that these user blocks will be mostly read and rarely written. So, Bondhu aims to optimize read latency (and thus read throughput).

We make one final point about disk geometries. Although they're often proprietary, manufacturers do present a logical abstraction of the disk called the logical block addressing (LBA) scheme. This is a simple linear addressing scheme in which blocks are addressed by an integer index starting from 0. The LBA scheme is essentially a one-dimensional representation of the disk's complex physical geometry. Disk manufacturers ensure that accessing consecutive blocks in the LBA space is similar to accessing consecutive blocks in the physical geometry.

Disk Layout Algorithm

Bondhu adopts an approach to disk layout algorithms for OSN applications that takes the community structure into account. This approach has multiple benefits. First, the problem complexity is reduced — that is, when making a disk placement decision inside a community, we can consider only that community's members. In addition, a bad placement choice will have relatively less impact because its effects will likely be limited by the community boundary. Finally, we can use existing community-detection techniques that are known to find good-quality communities in a social graph.

Bondhu's disk layout approach is presented in Figure 2. Its disk layout algorithm consists of three modules: *community detection*, *intracommunity layout*, and *intercommunity layout*.

Community Detection

The community-detection module organizes users in the social graph into clusters, so many edges connect users within the same cluster while relatively few edges connect users across clusters. The module uses existing *graph partitioning* and *modularity optimization* techniques. We chose these two algorithm classes because they operate on graphs with many vertices, they produce good clusterings, and they're fast — that

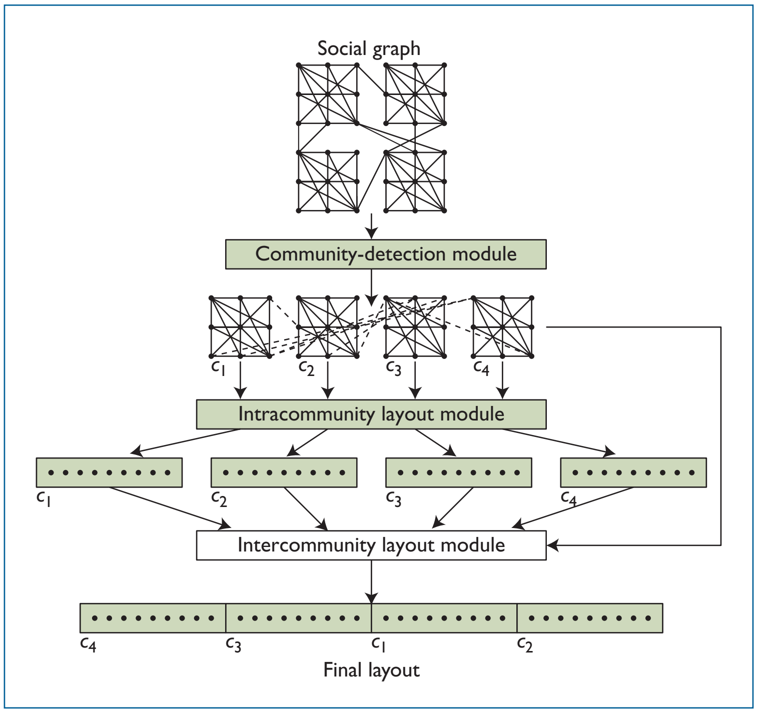

Figure 2. The Bondhu system's approach to disk layout. It consists of three modules: community detection, intracommunity layout, and intercommunity layout.

is, they find communities on graphs containing millions of nodes within seconds.

Graph-partition-driven community detection. Our graph-partition-driven community-detection algorithm (ParCom) is based on the multilevel k-way hypergraph partitioning scheme.[4] ParCom partitions the social graph into k equal subsets to minimize the edge cut. This is equivalent to minimizing the number of friends in other partitions. ParCom first coarsens the social graph into a small number of vertices; it constructs a sequence of smaller graphs from the original graph by collapsing vertices together using the heavy-edge matching (HEM) technique. ParCom also recalculates the edge weights. Then, it divides this smaller graph into k parts using a recursive bisection scheme. Finally, it uncoarsens the partitions back to the original graph in steps, and refines them at each step using local refinement heuristics. Further details are available elsewhere.[4]

Modularity-optimization-driven community detection. Our modularity-optimization-driven community-detection algorithm (ModCom), based on existing techniques,[5] detects good-quality communities in large networks (118 million nodes).

A partition's modularity is a scalar value between –1 and 1 that measures the density of intracommunity links as compared to intercommunity links. More specifically, modularity is the fraction of edges that fall within the communities minus the expected value of that fraction if the edges were randomly distributed (by preserving node degrees). Formally, we define this as

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

where M is the number of edges; $\delta(V_i, V_j)$ is the weight of the edge between users V_i and V_j; k_i is the degree of user V_i (the sum of the link/edge weights connected to user V_i); c_i is the user V_i's community; and $\sigma(c_i, c_j) = 1$, if $c_i = c_j$, and 0 otherwise.

ModCom works in two phases. First, it arranges users in a random order and assigns each one to a different community. For each user V_i, ModCom calculates the gain in modularity by removing the user from its own community and assigning it to any of its friends' communities. ModCom then moves user V_i to its friend V_j's community, for which the modularity gain is maximum. If there is no modularity gain, V_i stays in its original community. This first phase repeats iteratively for all users until no further gain in modularity is possible. In the second phase, ModCom creates a new graph consisting of the communities obtained in the first phase. It recalculates the edge weights for this phase, then runs the first phase again, continuing the process until no further changes are possible. More details are available elsewhere.[5]

Intracommunity Layout

Next, the intracommunity layout module takes as input the communities that the community-detection module produced, and creates a layout for the users within each community. We use a greedy heuristic to find a layout for a community. The heuristic starts with the most popular user – that is, the user with the highest edge degree (the sum of the link weights) – and places that user's block in the middle of the disk layout. Next, among all that user's friends, we choose the one connected to the user via the heaviest edge. This ensures that if two users are strongly connected, they'll be placed near each other on the disk. In the event of a tie, we choose the friend with the higher edge degree — that is, the more popular friend. Intuitively, by adding a popular user early, we provide more choices for the greedy algorithm later. We place this friend to the left of the already placed user on the layout. We then create a modified graph by merging the user and his or her friend. The edges connected to these two users are now connected to the combined node. For common friends 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 of this combined node's friends, we choose the one that's connected to it with the heaviest edge and place it to the right. We repeat these steps by alternately and iteratively placing the friends to the left and to the right of the already placed users. This alternating approach aims to preserve as much locality in the OSN nodes and edges of high weight as possible on the linear disk.

Intercommunity Layout

The intercommunity layout module takes as input the communities the community-detection module has produced and the layout the intracommunity layout module has produced within each community. It then creates a layout of the communities, letting us capture the relationships among them. For example, if a community c_i is strongly connected to another community c_j, we should place them close to each other on the disk, for reasoning similar to that used for the intracommunity layout module.

To create the intercommunity layout, we create a graph using the different communities as vertices. We calculate the weight of the edge between community c_i and community c_j as the sum of the edge weights from the members of community c_i to the members of community c_j. After creating the community graph, we run the same alternating and iterative algorithm we ran with the intracommunity layout module, but with communities instead of nodes. This gives us a layout of communities. Finally, we expand the layout within each community to obtain the OSN's final layout.

Modeling OSN Dynamics

Because of the scarcity of extensive open traces for OSN dynamic operations, we present three realistic models to capture OSN user interactions. These models differ in how they assign weights to the edges between users.

The *uniform model* is the simplest of the three. Here, 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 his or her friends' information. If the only operation allowed in the OSN were listing the names of all of a given user's friends, this model would be the best match.

The *preferential model* is motivated by the observation that a user with more friends will likely be more active than one with fewer friends — he or she will make more status updates, post more frequently, and so on. In addition, while browsing, users are more likely to access their more active friends' information.

To capture this type of interaction, the weight of the edge (V_i, V_j) should be proportional to V_j's edge degree. This metric isn't symmetric — that is, if V_j has a higher edge degree than V_i, then the weight of (V_i, V_j) is higher than that of (V_j, V_i). On the other hand, disk locality is symmetric in nature. So, 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 our disk layout algorithm.

The *overlap model* is motivated by the following observation: two users with more common friends are more likely to share interests and access each other's information than two users with fewer common friends. In other words, if user V_i has p friends in common with V_j and q friends in common 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 don't assign a 0 weight to the edge (V_i, V_j) if there are no common friends — an edge weight of 0 indicates that no edge exists at all.

Implementation and Evaluation

We implemented Bondhu as a layout manager for the Neo4j graph database, which is suitable for building OSN applications because it offers a graph-oriented model for data representation. A Neo4j graph consists of nodes, relationships, and properties. Properties map 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 modified Neo4j's PropertyStore so that the records are organized according to Bondhu's layout algorithms. We rely on the native file system so that our layout decisions are propagated to the disk block level — that is, the modified PropertyStore database Bondhu produces is stored sequentially on the disk. So, we start with an empty disk and verify with the *davl* tool (http://davl.sourceforge.net) that the database file is stored sequentially on the disk at the block level.

We implemented Bondhu in Java. The community-detection module uses the METIS library (http://glaros.dtc.umn.edu/gkhome/views/metis) for the ParCom algorithm.

We performed trace-driven experiments with our Bondhu implementation in Neo4j. We used the Facebook New Orleans dataset,[6] which contains 63,731 users and 817,090 links. We assigned appropriate weights to the social graph according to our uniform, preferential, and overlap models. Because our experiments suggest that preferential and overlap models yield similar numbers to the uniform model, our plots use the uniform model unless otherwise specified. Also, due to space constraints, we omit experimental results for the ModCom algorithm, which performs worse than the ParCom algorithm. This is because the uneven communities ModCom produces are suboptimal in minimizing the intercommunity cost.

We use two metrics to evaluate Bondhu's data layout schemes. The first is the cost as defined in Equation 1. This measures the spatial clustering of a user's friends on the disk. A lower cost means operations such as friend listing and wall posting will be faster. Our second metric is (user-perceived) response time. This measures the time required to fetch data blocks from all of a random user's friends. A low response time also means that the disk can handle more requests per unit time.

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 for each user in the graph. Bondhu handles the data layout automatically beyond this point. We next write an automated script that logs into the system as a random user and fetches the data blocks for all that user's friends (the "list all friends" operation). We measure the response time for the whole operation. To ensure that it isn't adversely affected by other processes accessing the disk at the same time, we carry out the same operation six times and use the minimum for the plots. We repeat this for 1,500 random users. We use this workload for all our experiments unless otherwise specified.

To ensure that the data is served from the disk (and not from the 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.

We conducted all our experiments on an HP DL160 compute block with two quad-core Intel Xeon 2.66-GHz processors, 16 Gbytes of memory, and 2 terabytes of storage.

Visualization of Block Access Patterns

Before evaluating Bondhu, we visually compare the disk block access patterns of Bondhu-enabled Neo4j against default Neo4j. We trace the disk blocks each query accesses using the *blktrace* tool, and use the *seekwatcher* tool to visualize the disk block access over time. We use the same workload as discussed earlier, with data block size per user set to 400 Kbytes. Figure 3 shows disk block access patterns under this workload (first 186 seconds) for default Neo4j.

Each dot in the figure depicts a block access at a particular location on the disk at a particular time. Because the user's friends aren't clustered on the disk, block accesses at a point in time are scattered all over the database. This effect is prominent when users with many friends are issuing the queries — for example, around 23 seconds and 46 seconds. So, the disk head incurs significant seek time for this query, which leads to a high response time.

Next, we visualize disk block access patterns under the same workload for Bondhu-enabled Neo4j in Figure 4. Here, we use ParCom with 64 communities. We observe a significantly better (sparser) disk block access pattern here than in Figure 3. Additionally, in Figure 3, the block accesses are scattered, whereas in Figure 4, they're clustered (prominent at 23, 46, and 116–139 seconds). This suggests that Bondhu is clustering the related friends' data near each other

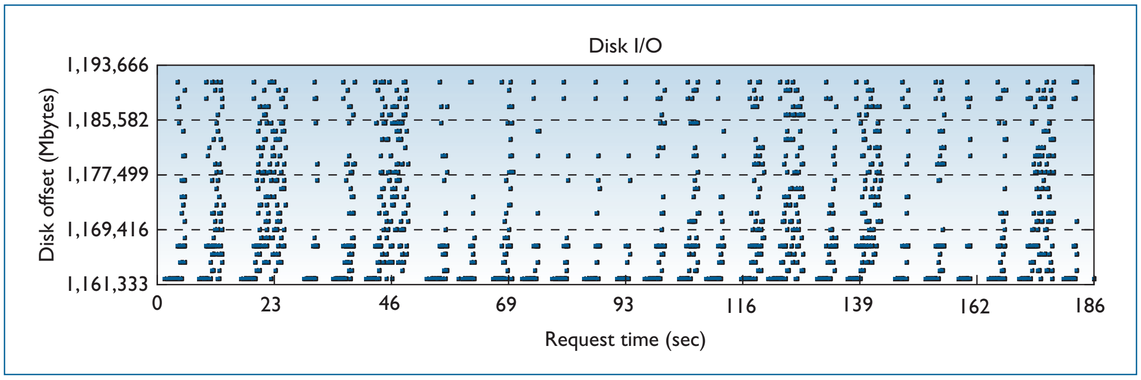

Figure 3. Blocks accessed in Neo4j for a "list all friends" query. Each dot depicts a block access at a particular location on the disk at a particular time.

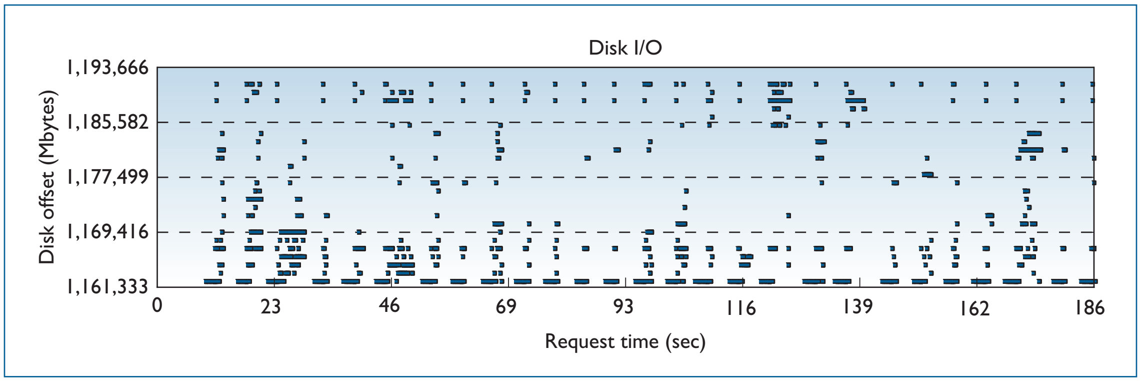

Figure 4. Blocks accessed in Neo4j with the Bondhu system handling data layout. Each dot shows a read request, its disk offset, and the time of the request. We compare these results with those in Figure 3 (the default layout).

on the disk. This translates to less disk arm movement and thus faster seek and response times.

Data Size

In an OSN application, the data associated with a particular user (name, address, profile picture, wall posts, and so on) can vary in size. So, it is important to measure the effect of varying user data block sizes on the layout algorithms' performance. Here, we first examine the effect of varying data sizes on the response time metric. Later, we present a scatter plot to show the correlation between the cost metric and response time improvements.

We vary data block size from 4 bytes to 4 Mbytes for each user and conduct our workload-based measurement. We use ParCom with 64 communities, compare it with the default layout, and plot the improvement in response time (the lower the response time compared to the default layout, the greater the improvement). We plot the 5th percentile, quartiles, and

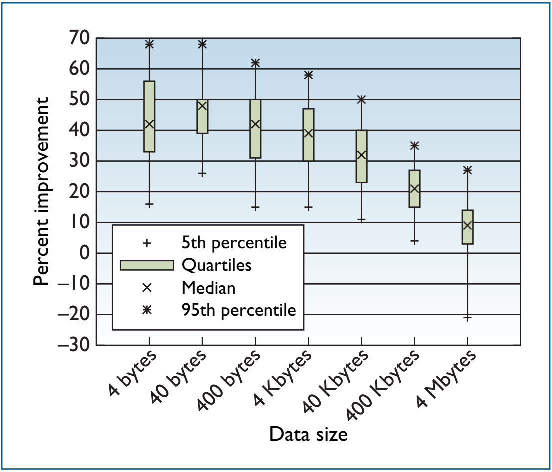

Figure 5. Improvement in response time without caching. We compared Bondhu's performance to that of the default layout for various data sizes.

the 95th percentile of the improvement for different data sizes.

As Figure 5 shows, Bondhu performs best when the user data size is 40 bytes. When data

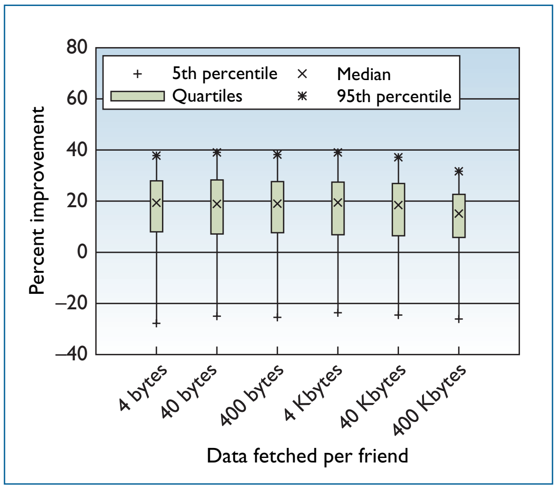

Figure 6. Improvement in response time when block sizes are different across users (without caching). We compared our results to that of the default layout.

size increases from 4 to 40 bytes, the median improvement increases from 42 to 48 percent. Beyond that, the improvement percentage decreases, and at 4 Mbytes, it reaches 9 percent.

This occurs for the following reasons. The native file system reads data in 4-Kbyte blocks. So, when user block sizes are small, a file system read fetches multiple users' data together. For example, with a 4-byte user block size, a read yields data for around 1,024 users. With a 40-byte user block size, a read yields data for around 102 users, and with a 400-byte user block size, it yields data for around 10 users. Because of the default layout's random data placement scheme, the expected number of friends present per read decreases by a factor of 10 when data block sizes grow from 4 to 40 to 400 bytes. Bondhu's clustering property prevents the expected number of friends present per read from decreasing much when data block sizes grow from 4 to 40 bytes. When the block sizes grow from 40 to 400 bytes, however, the expected number of friends present per read decreases dramatically. Thus, we see the drop in improvement after 40 bytes.

So, when user data size is smaller than the file system block size, the improvement is high because a single file system read yields more correlated data, reducing the number of seeks required to fetch all friends' data. Once user data sizes grow larger than the file system block size, Bondhu improves performance by reducing the seek distance. However, because fetching 4 Mbytes of data per friend at a time is highly impractical in an OSN, we repeated the experiment by fetching 4 Kbytes of data per friend when the per-user data block size is 4 Mbytes and observed a median improvement of 20 percent. This suggests that for large data blocks, the actual data transfer time dominates over the seek time and rotational latency. Hence, the decline in response time improvement.

Because the most recent data in an OSN is the most accessed data, these experimental results suggest that we should store each user's most recent data in small-to-medium data blocks. Our analysis of Facebook profile information shows that a 40-Kbyte data block is sufficient to store a user's profile information and picture, friend list, several recent wall posts, and a pointer to the older data.

Bondhu can organize the older data's layout as well. In this case, the amount of data stored per user will be different. To analyze how Bondhu performs for unequal block sizes, we store data proportional to the number of friends per user. Data block size varies from 512 Kbytes to 549 Mbytes according to a power-law distribution. We then select a random user and fetch X bytes each from all of his or her friends ($X = 4$ bytes, 40 bytes, 400 bytes, 4 Kbytes, 40 Kbytes, and 400 Kbytes). This corresponds to fetching a fraction of data from each friend to construct a newsfeed for that particular user. Figure 6 shows the results. We observe up to 20 percent median improvement in response time across various data sizes.

Next, we examine the correlation between the cost metric and response time improvements in Figure 7. This demonstrates how Bondhu's smart placement decision translates to better application-level performance. As we defined previously, a user's cost metric is the sum of differences between that user and his or her friends' data location. We calculate users' cost metric using the placement in both the default layout and the ParCom layout. A larger cost denotes that a user's friends are far away in the disk. We then calculate the fraction of improvement in the cost metric using Bondhu's ParCom layout scheme over the default layout. For the corresponding users, we measure the fraction of improvement in the response time metric and plot the results. Figures 7a and 7b show the results for two different data sizes. Figure 7a shows good correlation because most points are along the $x = y$ line. For Figure 7b, the correlation is less prominent owing to the reasons discussed earlier.

Caching

In the previous experiments, all requests were served from the disk and not the memory. However, because serving results from memory reduces response time significantly, we enable caching for both Neo4j and Bondhu and examine the effect on the response time metric.

We use the same workload as earlier but without flushing the cache between successive user requests. A user issues 10 successive requests to fetch all his or her friends' data blocks. As before, we conduct this experiment for 1,500 randomly selected users.

As with the previous experiment, we plot the 5th percentile, quartiles, and the 95th percentile of the improvement for the different data block sizes (see Figure 8). When the data size is small, we don't see much improvement using our layout scheme (the decline at 400 bytes is due to the operating system's caching behavior). As the data sizes increase from 4 to 40 to 400 Kbytes, Bondhu's benefits kick in, as the rise in median response time improvement demonstrates. When the data sizes are small, all user information can be kept in memory. So, 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 (40 Kbytes in this case), the user data blocks can't all be kept in memory, but must be fetched from disk, so the behavior we described in the previous section kicks in.

In summary, Bondhu's worst-case median improvement is 0 percent (small data sizes with caching), and the best case improvement is 48 percent (medium data sizes without caching). So, it's always preferable to use Bondhu as the disk layout manager.

Number of Communities

In the ParCom algorithm, we can tune the number of communities. The fewer the number of communities, the larger a given community's size. For instance, with only a single community, the intracommunity layout module handles the layout decision solely. As the number of communities increases, the intercommunity layout module gains influence on the layout decisions. For a given social network graph, we want to tune the number of communities such that we obtain the best disk layout.

We examine cost metric improvements by varying the numbers of communities in ParCom.

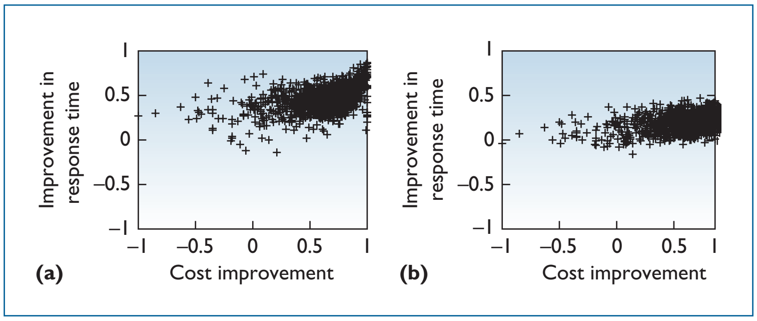

Figure 7. Correlation between cost and response time improvements (without caching). We measured the correlation using (a) 40-byte and (b) 400-Kbyte blocks.

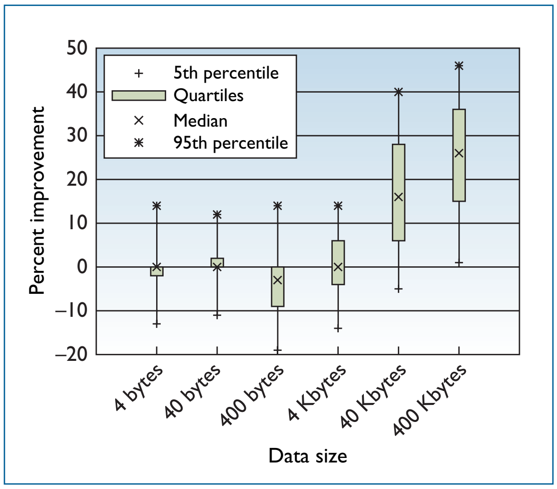

Figure 8. Percent improvement in response time with caching enabled. We compared Bondhu's performance to that of the default layout for various data sizes.

We use the same workload discussed earlier, with a data size per user of 4 Kbytes. As Figure 9 shows, as the number of communities increases from 2 to 32, the cost metric steadily improves. With fewer communities, the intracommunity-detection module is mostly responsible for the layout, and Bondhu doesn't capture the social graph's community structure. So, the improvement grows quickly as the number of communities increases. However, this curve hits a knee at 64 communities and plateaus out thereafter because the community-detection module has lower marginal utility in finding more community structure in the graph toward the right end of the plot.

Different Models

Thus far, all the experimental results are based on the uniform model. Here, we present results using all three models. To provide a baseline for

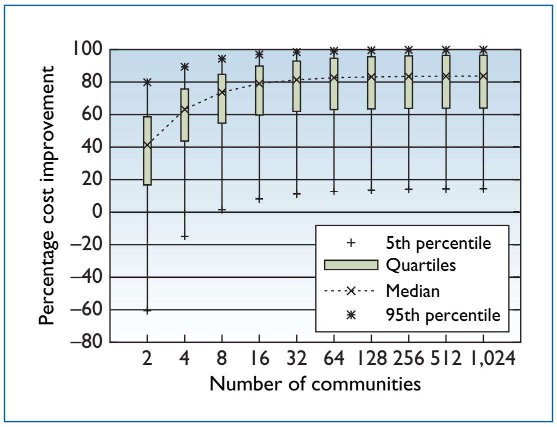

Figure 9. ParCom performance. We varied the number of communities and examined the improvement in the cost metric.

comparison, we also present results using the default Neo4j layout. We use the same social graph as in the previous experiments. The data block size for each user is 4 Kbytes. Bondhu takes the model as input, creates a layout using that model, and organizes the data according to that layout.

We use three different workloads based on the graph models. First, in the uniform workload, we randomly select a user who issues a request to access a friend's blocks at random. Second, in the preferential workload, the randomly selected user issues a request to access a friend's data blocks with probability proportional to the friend's degree. Third, in the overlap workload, the randomly selected user issues a request to access a friend's data blocks with probability proportional to the number of friends the user and friend have in common. In each workload, the user issues 1,000 successive requests, and we measure the response time. We take each measurement 10 times and use the minimum response time. We conduct this experiment for 1,000 users in total.

Figure 10 shows the results of running each workload on the four layouts. We denote each run of the experiment as model:workload, where model denotes the models we use (preferential, overlap, uniform, or default) and workload denotes the workloads we use (preferential, overlap, or uniform). We plot the 5th percentile, quartiles, and the 95th percentile of the response time.

We can make three observations from this plot. First, the default layout performs twice as badly as any of the other models (with a median response time of 175 ms). Second, the performance of the layout produced by the uniform model is comparable to the performance of the layouts produced by the preferential and overlap models. Third, a specific layout's performance doesn't vary much over the different workloads.

To investigate whether the choice of users affects the results, we first sorted the users according to their number of friends. We then repeated the experiment with the 20th and the 80th percentiles of users from the distribution. The results are statistically the same — that is, the preferential and overlap models don't show significant improvement over the uniform model. We can attribute this first to OSNs' strong community structure — the uniform model appears sufficient to capture this structure. Second, because the communities are coarse-grained (1,000 users per community), using the different models doesn't change community membership significantly. This contributes to invariability across different models.

One directional conclusion we can take from these observations is that although we can create complex models[7] to capture user interactions in a social network, the simplest model is often sufficient for making important disk layout decisions. Taking more complex models into account might yield little added benefit for the amount of effort involved. The social graph

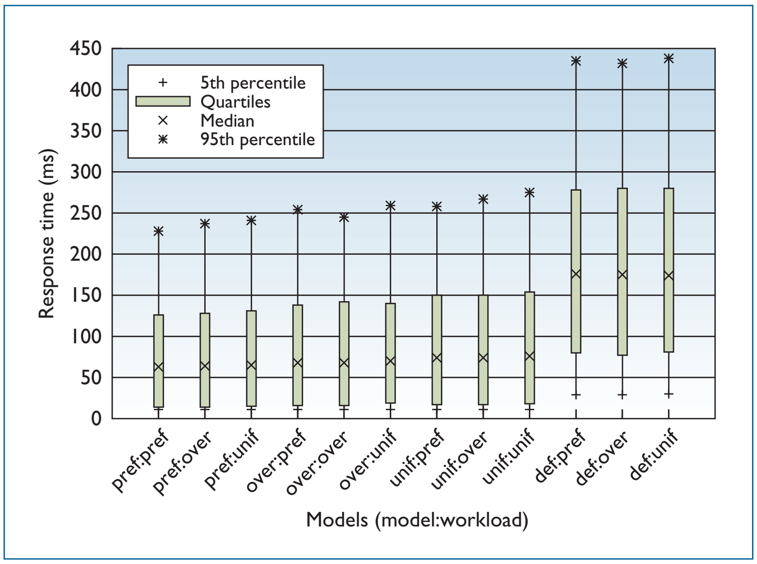

Figure 10. The effect of different models on response time. We measured response time for three workloads (preferential, overlap, and uniform) running on four different layouts (preferential, overlap, uniform, and default).

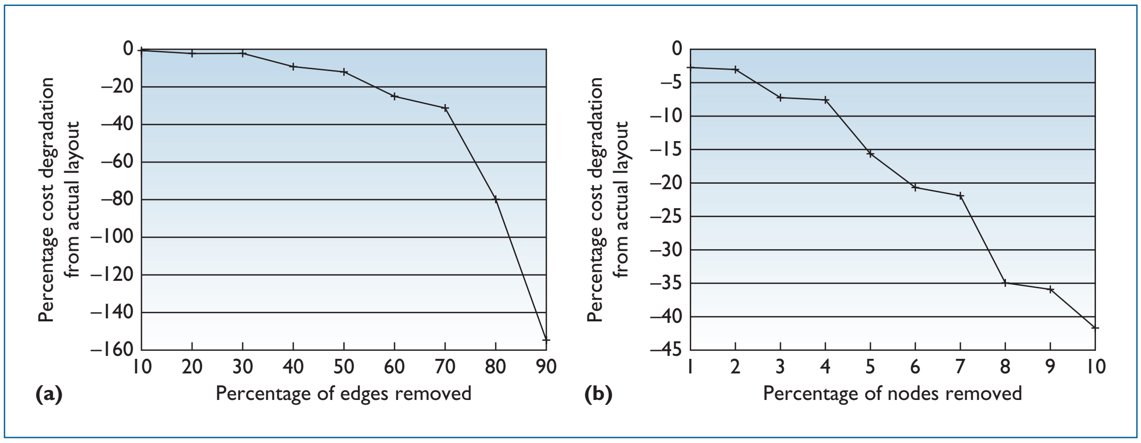

Figure 11. The effect of online social network evolution on older layout performance. We measured this effect with (a) edge removal and (b) node removal.

structure has the biggest influence on disk performance.

OSN Evolution

As an OSN evolves, we might need to reorganize the layout to reflect the new social graph. Although techniques for modifying disk layout incrementally are beyond this article's scope, we examine how frequently Bondhu should reorganize the layout. To do this, we first create older versions of the graph by removing edges and nodes at random. Next, we use the layout obtained from the older version to host the latest version of the graph. We place the nodes in the latest graph that aren't present in the older graph sequentially after the old layout to produce the new layout. We measure the percentage of degradation in the cost metric from using the older layout instead of the layout based on the latest graph. Intuitively, as long as the percentage of degradation is reasonable, we can use the old layout, and reorganization is unnecessary.

Figure 11a shows that even when the layout is based on a graph with 40 percent of its edges removed from the current graph, the cost metric degrades by less than 10 percent. Most OSNs reach a plateau in terms of the number of nodes after a point. So, reorganization will be infrequent after this point. On the other hand, Figure 11b shows that when the layout is based on a graph with only 5 percent of the nodes removed from the current graph, the degradation is greater than 15 percent. Note, however, that an old layout's performance can only be as bad as the default layout. Thus, although a layout based on 10 percent fewer nodes has 42 percent cost degradation compared to the layout based on the current graph, it still has around 72 percent cost improvement over the default layout (because the layout based on the actual graph has 80 percent cost improvement over the default layout; see Figure 9). Facebook grew roughly 50 percent over nine months (March 2009 to December 2009; http://tinyurl.com/4flny8c). Thus, even at this growth rate, reorganization can occur infrequently and still do better than the default layout. Runtime reorganization of the disk layout can be performed efficiently using existing techniques.[8]

Although solid state disks (SSDs) are becoming increasingly viable alternatives to disks for data storage, we believe that disks won't go away. Due to write lifetime issues, for instance, disks are often used as a cache for SSDs.[9] In addition, disks will also likely be cheaper per byte for several years. Thus, finding solutions for improving disk performance will remain imperative.

Deploying a Bondhu-enabled social network in the real world can advance this research direction. We believe that incorporating Bondhu into real deployment scenarios can open up further research directions such as exploring the interplays between frequently modified data and Bondhu, between indexing and disk placement, and between querying and data placement optimization.

Acknowledgments

This work has been supported in part by US National Science Foundation grant CCF 0964471, and in part by the Air Force Research Laboratory/Air Force Office of Scientific Research Assured Cloud Computing Center.

References

1. J.M. Pujol et al., “The Little Engine(s) that Could: Scaling Online Social Networks,” *Proc. ACM SIGCOMM 2010 Conf.* (SIGCOMM 10), ACM Press, 2010, pp. 375–386.
2. M.R. Garey, D.S. Johnson, and L. Stockmeyer, “Some Simplified NP-Complete Problems,” *Proc. 6th Ann. ACM Symp. Theory of Computing* (STOC 74), ACM Press, 1974, pp. 47–63.
3. J. Petit, “Experiments on the Minimum Linear Arrangement Problem,” *J. Experimental Algorithmics*, vol. 8, Dec. 2003, pp. 2.3:1–2.3:29.
4. G. Karypis and V. Kumar, “Multilevel *k*-way Partitioning Scheme for Irregular Graphs,” *J. Parallel and Distributed Computing*, vol. 48, no. 1, 1998, pp. 96–129.
5. V.D. Blondel et al., “Fast Unfolding of Communities in Large Networks,” *J. Statistical Mechanics: Theory and Experiment*, vol. 2008, Oct. 2008, p. P10008.
6. B. Viswanath et al., “On the Evolution of User Interaction in Facebook,” *Proc. 2nd ACM Workshop Online Social Networks* (WoSN 09), ACM Press, 2009, pp. 37–42.
7. F. Benevenuto et al., “Characterizing User Behavior in Online Social Networks,” *Proc. 9th ACM SIGCOMM Conf. Internet Measurement* (IMC 09), ACM Press, 2009, pp. 49–62.
8. B. Salmon et al., “A Two-Tiered Software Architecture for Automated Tuning of Disk Layouts,” *Proc. Workshop Algorithms and Architectures for Self-Managing Systems*, ACM Press, 2003, pp. 13–18.
9. G. Soundararajan et al., “Extending SSD Lifetimes with Disk-Based Write Caches,” *Proc. 8th Usenix Conf. File and Storage Technologies* (FAST 10), Usenix Assoc., 2010, pp. 101–114.

Imranul Hoque is a PhD student at the University of Illinois, Urbana-Champaign. His primary research interests include designing, implementing, and evaluating techniques to improve the performance of new classes of storage systems, such as graph databases and key-value storage systems. Hoque has a BSc in computer science and engineering from the Bangladesh University of Engineering and Technology. Contact him at ihoque2@illinois.edu.

Indranil Gupta leads the Distributed Protocols Research Group in the computer science department at the University of Illinois, Urbana-Champaign. His interests include distributed protocols, large-scale distributed systems, monitoring and management for distributed systems, and sensor networks. Gupta has a PhD in computer science from Cornell University. He’s a recipient of the US National Science Foundation’s CAREER award. Contact him at indy@illinois.edu.