Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph

Rui Wang Christopher Conrad Sam Shah
LinkedIn

Abstract

Social networks often require the ability to perform low latency graph computations in the user request path. For example, at LinkedIn, we show the graph distance and common connections when we show a profile in any context on the site. To do this, we have developed a distributed and partitioned graph system that scales to hundreds of millions of members and their connections, handling hundreds of thousands of queries per second.

To accomplish this scaling, real time distributed graph traversal is converted into set intersections that are accomplished in a scatter/gather manner. A network performance bottleneck forms on the gather node as it must merge partial results from many machines. In this paper, we present a modified greedy set cover algorithm that is used to locate the minimal set of machines that can serve the partial results. Our results indicate that we are able to save 25% in the 99th percentile latency of these graph distance calculations for LinkedIn's social graph workloads.

1 Introduction

Many online social networks require the ability to perform graph computations in the request/response loop. Nowhere is this more acute a problem than at LinkedIn, the largest online professional social network with 225 million members and a company founded around the notion of members and their networks.

For example, Figure 1 shows the search results page. Here, the degree distances (1st, 2nd, 3rd) and the number and set of common connections to the searcher are shown. For user experience reasons and because these are core to the concept of the LinkedIn product and its notion of a professional network, this decoration cannot be approximated. For example, visibility and privacy business rules require that members cannot see profiles more than three degrees away from them in the social graph.

At LinkedIn, these graph-based metrics are computed online with our distributed graph service. This service allows application developers to perform common graph operations such as retrieving a member's connections, computing shared connections between members, and calculating distances in the graph (that is the graph distance)—all on page load. The system easily supports graphs of hundreds of millions of nodes. There is considerable request response fan-out from applications—for example, the search result page performs one common connection and one degree distance calculation per result—and this distributed graph serves hundreds of thousands of queries per second. Naturally, for a good user experience, latency is our primary concern.

The graph service is horizontally partitioned so that a member's entire adjacency list is stored on one physical node. Computing the graph distance in a naïve fashion through breadth-first traversal is intractable as it would require $O(n^2)$—where n is the average number of connections per member—distributed calls throughout the cluster, effectively touching every machine. Instead, a two-tier architecture is employed. The source's second degree is stored as compressed sorted arrays in the *Network Cache Service (NCS)*, which communicates with a key-value database of edges called *GraphDB*. Graph distances are computed through efficient set intersections. Around 80% of the calls for distances at LinkedIn can be satisfied by NCS and thus generate no additional remote calls to GraphDB.

However, a challenge arises during a second degree cache miss. To construct a member's second degree array, scatter-gather merging occurs that deduplicates partial second degrees from each GraphDB node. As social networks are tightly interconnected, there is significant overlap in this set merge. The merging can be performed at either the NCS layer, which creates a bottleneck on bandwidth [9] and CPU resources, or preferably can be pushed down to the GraphDB nodes as much as possible. To maximize such merging, we want to find the optimal set of GraphDB nodes that can serve such second degree queries to reduce merging within NCS. In this paper, we

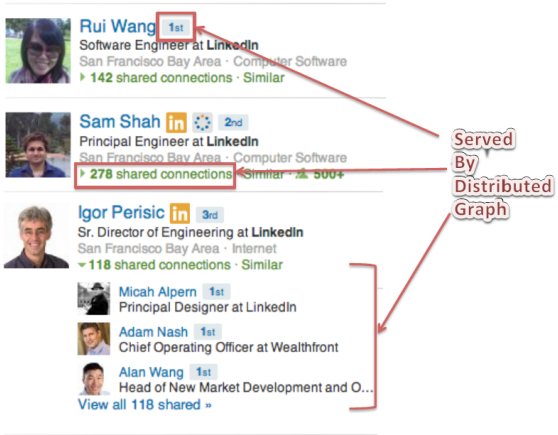


Figure 1. An example search result page showing the top 3 results. Each result is decorated with the number and set of connections in common and the degree distance from the searcher (1st, 2nd, or 3rd) as indicated by the picture.

will explain how we use a greedy set cover algorithm to reduce the average number of GraphDB nodes requested to serve second degree queries, and to move merging and deduplication from NCS to GraphDB nodes.

Results with LinkedIn's graph and workload indicate that this optimization reduces 95th and 99th quantile cache construction latency by 18% and 38% respectively, cumulating in a 25% reduction in the 99th percentile latency for graph distance calculations. This latency reduction is particularly important because it reflects a better user experience for members with a large number of connections—LinkedIn's most active members.

While this paper presents LinkedIn's solution, the authors believe large-scale low latency graph queries are an emerging research direction. This problem presents significant challenges around scalability, particularly as real-world graphs exhibit power law distributions [1].

2 The Distributed Graph

LinkedIn's distributed connection graph infrastructure consists of three major subcomponents as shown in Figure 2. Subcomponents include graph database, GraphDB, which stores member connections; a consistently hashed [5] caching layer that converts distributed graph traversals into efficient set operations; and an API layer that isolates the frontend clients from graph query implementations. It handles arbitrary graphs, including but not limited to the social graph. For ease of presentation, we will focus on the social graph, but without loss of generality, these techniques apply to graphs of other entities in the LinkedIn ecosystem as well.

LinkedIn's distributed graph supports the following APIs:

GetConnections — Returns a member's connections based on a member ID and optional filters. This API answers questions such as "Who does member X know?" and "How many new connections did member Y add since the last log in?".

GetSharedConnections — Compares two members' connections and returns the intersection. This API answers questions such as "Who do I know in common with member Y?".

GetDistances — Includes more than 50% of the graph operations. It takes a source member ID and a set of destination member IDs, then returns the network distance between the source to each destination up to three degrees of separation.

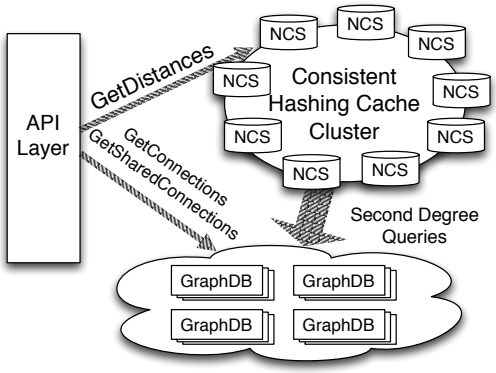


Figure 2. LinkedIn's distributed graph system consists of three major subcomponents: a partitioned and replicated graph database referred to as GraphDB; a distributed cache called Network Cache Service (NCS) that stores a member's network and serves queries requiring second degree knowledge; an API layer for the frontend.

2.1 Graph Partitioning and Replication

In GraphDB, members are represented as vertices, and connections as edges. It partitions this graph into a cluster of physical machines based on member IDs. A hash-based partitioning is used so that workloads are more likely to be evenly distributed. Partition IDs are calculated based on member IDs, eliding the need to maintain a heavy in-memory data structure of a mapping between member IDs and partition IDs. Each member's connections are sorted by ID and colocated on the same partition that stores their information. We use N to represent the total number of partitions in our system. N is chosen to be large enough so that when the graph grows, the partition size remains reasonably small so as to fit several onto one physical machine.

Each partition is replicated on at least R different machines to provide failover and load balancing. Each physical machine holds P partitions. If we use Z to represent the total number of physical machines in the cluster, then $Z = (N \cdot R)/P$.

To prevent hot spots, we randomize the set of partitions colocated physically. If two partitions are stored on the same machine in one replica, they will be stored on differ-

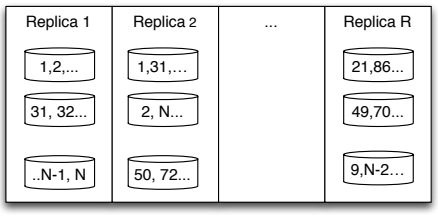


Figure 3. Partitions are randomly grouped on physical machines in each replica. We have a total of N partitions. Each node stores P partitions, thus one replica consists of $\lceil \frac{N}{P} \rceil$ nodes. Replication factor is R. The entire cluster consists of $(N \cdot R)/P$ nodes.

ent machines in other replicas. This ensures that if both partition i and partition j are more frequently visited, they are only colocated on the same physical machine once.

For design and operational simplicity, we do not distinguish which replica a machine belongs to once it joins the cluster. At LinkedIn, the entire cluster of machines storing member's first degree connections and its replicas is called GraphDB. In this paper, each physical machine in this cluster is referred to as a *GraphDB node* or simply as a *node*.

2.2 GetDistances Algorithm

Both GetConnections and GetSharedConnections have a complexity of $O(n)$ where n is the average number of connections per member. In both cases, the API layer fetches connections from GraphDB once and performs any filtering or intersection locally.

GetDistances requires multiple hops. With Breadth First Search (BFS), cost complexity increases exponentially: looking up a member connection's connections in the worst case would involve $O(n)$ remote calls with a cost complexity of $O(n^2)$. For most LinkedIn users, this call would cause one remote call to every GraphDB node in the cluster. We need a more efficient solution.

If we store a member's first and flattened second degree connections in a cache, then the distances between this member and a set of destinations can be determined by intersecting the destination IDs with the data in the cache. No call to GraphDB is required if all destinations are within two degrees. For those who fall out of two degrees, one remote call is generated per destination to fetch the connections and intersect them with the source member's second degree to determine whether they are three degrees apart.

This motivated us to build a caching layer, referred to as the *Network Cache Service (NCS)* to store the first and second degree connections of members visiting LinkedIn. This cache entry can then be used to perform additional GetDistances calls for the same user much faster during any active login session.

2.3 Network Cache Service

The API layer diverts GetDistances calls to NCS. If NCS contains a valid cache for the requested source member, distances can be computed in NCS by performing array intersections. If the cache is not available, this request waits for the cache entry to be built online. We use an LRU caching strategy with a short TTL to provide a real-time view of the graph. These arrays are compressed using delta compression [12] to further reduce memory usage.

One of the biggest challenges we face is to assemble the second degree connections quickly in real time on a cache miss. Some LinkedIn members have millions of second degree connections, yet we still need to serve GetDistances quickly. A member's second degree size grows exponentially compared to first degree connections. Investing in hardware alone cannot catch up with the exponential growth of connection density. We need to examine software solutions to manage latency growth.

To compute a member's second degree connections, we retrieve first degree connections from a single GraphDB node, then shard them into multiple GraphDB nodes to query their connections. Each GraphDB node receiving the second degree query will look up and merge its local result, removing duplicates prior to sending the results back. NCS waits until all responses return from the GraphDB nodes before merging the results into a final sorted array before compression.

Merges and deduplications are done on both in GraphDB and on NCS. In GraphDB, merging and deduplication is processed in parallel, whereas a single NCS machine merges all intermediate results into a complete sorted array that can be both CPU intensive and time consuming.

We measured the time spent on constructing network cache for members with various network sizes and found the time NCS spent on merging and removing duplicated connections from partial results is always the bottleneck and tends to be worse for members with larger networks. We also noticed that for users with a few hundred connections, partial responses often come back from all $(N \cdot R)/P$ GraphDB nodes instead of N/P machines. This means for these users, each GraphDB node is only merging $\frac{n}{(N \cdot R)/P} = \frac{n \cdot P}{N \cdot R}$ connections locally, where n represents the number of connections for a member. NCS, on the other hand, is performing a $(N \cdot R)/P$-way merge. Thus, the bigger the cluster we have, the less work will need to be done on the GraphDB side in parallel, leaving the $(N \cdot R)/P$-way merge in NCS the single significant bottleneck during cache construction.

We sampled a set of members having tens of connections to tens of thousands of connections. We measured the time NCS spent on merging partial results from a single replica GraphDB cluster versus our multi-replica

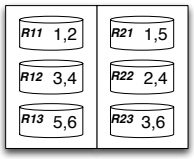


Figure 4. Simplified example graph: a distributed graph with six partitions. Node IDs are represented as Rij, where Ri1, Ri2, and Ri3 make one full replica of the graph. The integers inside the nodes are partition IDs.

production cluster. For users with a few hundred connections, time spent on merging results from the multi-replica cluster is twice as much as from a single replica. For users with tens of thousands of connections, the delta drops to a 50% difference.

If we minimize the fan-out and reduce the number of GraphDB nodes requested to serve second degree queries, we can reduce merges and deduplications done on the NCS side and take advantage of the performance gain mentioned previously. Our problem now becomes selecting a minimum number of GraphDB nodes that cover every partition requested in a member's second degree query. This an application of the set cover problem [3].

3 Set Cover in a Distributed Graph

Given a set of elements K=$\{1,2,...,m\}$ and $L = \{S_1, S_2, \ldots, S_i, \ldots\}$ whose union comprises K, the classic set cover problem is to find a minimum number of sets from L whose union contains all elements in K.

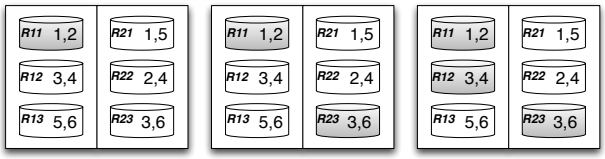
The set cover problem in our distributed graph is to find the minimal set of GraphDB nodes that covers K to serve a member's second degree query.

Set cover is known to be NP-hard, but a simple greedy heuristic algorithm offers a logarithmic ratio bound [3]. This algorithm works by picking, at each iteration, the set S_k that covers the most remaining uncovered elements in K. An example of how this works follows.

Assume we have a distributed graph comprised of six partitions. We store two partitions on each node with two replicas as shown in Figure 4. We use R11, R12, ... R23 to index the GraphDB node, thus L= $\{S_{R11}, S_{R12}, \ldots S_{R23}\}$.

A member's second degree connections are stored on partitions 1,2,3,4, and 6. To serve this query, this greedy algorithm first intersects $K = \{1,2,3,4,6\}$ with every set in L, then selects node $R11$ that covers the most partitions in K (Figure 5a). Partitions 1 and 2 are removed from K, and S_{R11} is removed from L. In the next iteration, each remaining set in L intersects with $K = \{3,4,6\}$, and node $R23$ is selected for maximum coverage. In the last iteration, $K = \{4\}$ and $R12$ is selected. See Figure 5 for illustration.

The ratio bound of this algorithm is $\ln(S_{max}) + 1$, where S_{max} is the maximum size of set S_i in L. When each GraphDB node stores P partitions, the ratio bound of greedy set cover in distributed graph becomes $\ln(P) + 1$.



(a) K=$\{1,2,3,4,6\}$ **(b)** K=$\{3,4,6\}$ **(c)** K=$\{4\}$

Figure 5. Using the classic greedy set cover algorithm to find a set of GraphDB nodes for second degree query. In 5a, K intersects with all 6 nodes and removes S_{R11} from L when it is done. In 5b, K intersects with each of the remaining 5 nodes, and removes S_{R23}. In the last step 5c, K intersects with the remaining 4 nodes from L.

$C \leftarrow \emptyset$
repeat
  $p_k \leftarrow$ randomly selected partition from K
  $nodes \leftarrow$ map$[p_k]$
  for node from $nodes$ not added to C **do**
    Find $node_k$ with coverage S_k maximizing $\mid K \cap S_i \mid$
  end for
  $K \leftarrow K - S_k$
  $C \leftarrow C \cup \{node_k\}$
until C covers all elements in K
return C

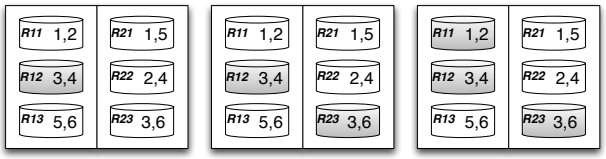
Figure 6. Modified set cover algorithm: traditional set cover algorithm performs $\mid K \cap S_m \mid$. S_m represents partitions covered by $node_m$, where $node_m$ represents all remaining unselected nodes; our set cover algorithm only performs set intersection between K and S_i, where $node_i$ contains some randomly picked partition ID p_k, dramatically reducing the number of set intersections required.

Without using set cover, we effectively have a ratio bound of R where R is the number of replicas in the system. In our system, R is almost twice as much as $\ln(P) + 1$. Applying greedy set cover algorithm, we are able to distribute the second degree query to 50% fewer GraphDB nodes, thus shifting the array merges from NCS to GraphDB nodes.

The catch is to find the set that has the maximum coverage during each iteration, so we intersect K with every remaining set S_i from L. We use l to represent the size of L. We will perform $O(l^2)$ set intersections. These intersections introduce an additional latency, significant for users with fewer than a hundred connections.

We are able to modify this greedy algorithm by taking advantage of an additional property of our sets: sets from the same GraphDB replica do not intersect with each other. The idea is to avoid doing an intersection between K and all remaining nodes by restricting it, such that it only intersects with a much smaller subset of nodes that's more likely to provide the best coverage.

We first built a mapping from partition IDs to GraphDB nodes. For partition p_i, map$[p_i]$ returns the set of GraphDB nodes that covers p_i.



(a) *nodes*=*R*12,*R*23 **(b)** *nodes*=*R*13,*R*23 **(c)** *nodes*=*R*11,*R*21

Figure 7. With modified greedy set cover algorithm, *nodes* represent the sets used in each iteration to intersect with K when partition 3 is selected in 7a and partition 6 is selected in 7b.

Our modified greedy algorithm is shown in Figure 6. Using the same example discussed previously, this modified greedy set cover algorithm will randomly pick a partition in $K = \{1,2,3,4,6\}$. If we assume partition 3 is selected, that means $nodes = \{S_{R12}, S_{R23}\}$. We now intersect these two nodes with K instead of every node in the cluster. If node $R12$ is selected, then $K = \{1,2,6\}$, and $C = \{R12\}$. In the next iteration, we randomly pick partition 6 from $\{1,2,6\}$, now $nodes = \{S_{R13}, S_{R23}\}$. Again only two intersections are required to pick the best coverage. If we say $R23$ is selected in this iteration, then $K = \{1,2\}$ and $C = \{R12, R23\}$. In the last iteration, no matter which partition is selected, $R11$ will be picked. See Figure 7 for illustration.

Instead of finding the set with maximum coverage by intersecting K with each remaining set from L, we have intersections only across the replicas covering the randomly picked partition ID p_k, thus reducing the $O(\lceil \frac{N}{P} \rceil \cdot R)$ intersection to $O(R)$. This implementation dramatically reduced the time spent on performing intersections, thus avoiding additional latency.

During each iteration, the algorithm guarantees that we either select two nodes that belong to the same replica, a node that offers equivalent coverage, or a node that removes at least the randomly picked partition p_k. Nodes that belong to the same replica as the already selected nodes have a higher probability of being picked because nodes from the same replica do not intersect with each other, and so have a better chance to offer higher coverage.

4 Evaluation

Our evaluation answers the following two questions:

1. What is the performance of NCS second degree cache creation with the set cover optimization?
2. What is the performance of the GetDistances call with the set cover optimization?

We ran these experiments on the LinkedIn graph service in production for one high traffic day by splitting the NCS nodes so that half ran the set cover optimization.

Figure 8 shows the 95th and 99th percentile latencies in building second degree caches. With the set cover optimization, the 95th percentile latency dropped 18% while the 99th percentile latency dropped 38%.

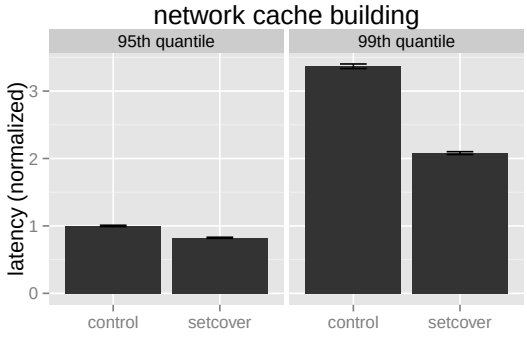


Figure 8. The latency of network cache building in NCS with and without the set cover optimization. The 95th quantile latencies dropped 18%, while the 99th latency dropped 38%.

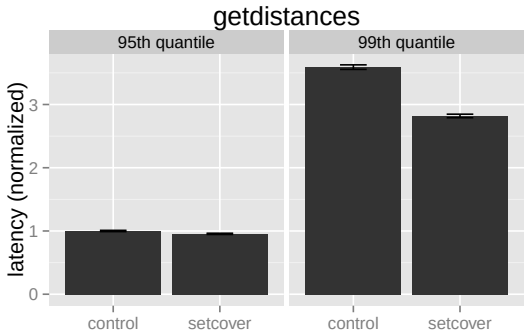


Figure 9. Latency of the GetDistances call with and without the set cover optimization. The 95th quantile latencies dropped 7% and the 99th quantile latencies are down 25%.

Figure 9 shows the 95th and 99th percentile latencies for the GetDistance call. The 95th percentile latency dropped approximately 7% while the 99th percentile latency dropped 25% on NCS nodes that use set cover optimization during request sharding.

In terms of aggregate cluster statistics, the total number of outbound traffic dropped over 40% while overall inbound traffic dropped 10%.

By reducing sharding and nodes visited in a second degree cache query, we moved work for second degree calculation from NCS to GraphDB nodes. NCS thus merges fewer responses. These results indicate that using a greedy set cover approach to find a close-to-optimal set of GraphDB nodes to serve second degree queries has a significant impact on reducing aggregate cluster bandwidth and overall latency of the system.

5 Related Work

Single-node graph databases are unable to scale to large graphs [11]. Most distributed graph systems research, such as Pregel [7] and GraphLab [6], focus on offline queries, particularly supporting iterative algorithms like PageRank. Their latencies are insufficient for any online serving application.

Research graph systems for online serving applications mostly focus on replicating social graph state to co-locate

a member's neighbors on the same physical node using one-hop replication strategies [10]. This reduces latency, but at a relatively high cost of additional replication. Recent work reduces this replication factor based on either read/write frequencies [8] or temporal locality [2]. Other research focuses on multicore or GPU-based mechanisms for graph traversal [4]. This research shows promise and some ideas can be incorporated into a real production system if the right level of performance, service, and operability can be achieved.

6 Conclusion & Future Work

In this paper, we presented an application of set cover to locate the minimal set of machines for merging partial second degrees in a distributed graph architecture. This insight allows us to reduce the 99th percentile latency by 38% in cache construction. As a result, we can achieve 25% reduction in the 99th percentile latency for graph distance calculations.

There is ample future work in this area. LinkedIn's distributed graph infrastructure stores not only quadratically growing members and their connections, but also other types of vertices and edges. For example, companies and members who follow these companies or schools and their alumni have orders of magnitude more edges and are significantly more skewed than the typical members. This requires new approaches.

For instance, we are investigating incremental updates to a member's second degree network without transferring and aggregating the entire dataset. The idea is, for each NCS node, the system calculates the second degree set for a member once, storing it in memory and persisting in second storage. When a user visits the site later, only incremental changes are fetched from GraphDB nodes based on the last timestamp stored. This approach can dramatically reduce the size of the data transferred, thus reducing burden on GraphDB and improving the second degree calculation time.

In addition, new site features require fairly complex queries that can identify advanced relationships between members, such as: member *A* who went to school with member *B* who worked years ago with member *C*—which must be satisfied in the request/response path. This increased data volume engenders further data distribution, increasing the penalty on remote communication for such queries. It also becomes unrealistic to build second degree caches for all or even a partial set of the combinations of entities in our graph stores.

References

[1] Lada Adamic and Bernardo Huberman. Zipf's law and the Internet. *Glottometrics*, 3:143–150, 2002.

[2] Berenice Carrasco, Yi Lu, and Joana M. F. da Trindade. Partitioning social networks for time-dependent queries. In *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, pages 2:1–2:6, New York, NY, USA, 2011.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Riverst. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1996.

[4] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 78–88, Washington, DC, USA, 2011.

[5] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC 1997*, pages 654–663, New York, NY, USA, 1997.

[6] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8): 716–727, April 2012.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J. C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD 2010*, pages 135–146, New York, NY, USA, 2010.

[8] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD 2012*, pages 145–156, New York, NY, USA, 2012.

[9] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*, pages 12:1–12:14, Berkeley, CA, USA, 2008.

[10] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine that could: Scaling online social networks. In *SIGCOMM 2010*, pages 375–386, New Delhi, India, 2010.

[11] Bin Shao, Haixun Wang, and Yanghua Xiao. Managing and mining large graphs: systems and implementations. In *SIGMOD 2012*, pages 589–592, New York, NY, USA, 2012.

[12] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar RAM-CPU cache compression. In *ICDE 2006*, Washington, DC, USA, 2006.