Distrib Parallel Databases (2017) 35:23–53
DOI 10.1007/s10619-017-7191-3

Distributed block formation and layout for disk-based management of large-scale graphs

Abdurrahman Yaşar[1] · Buğra Gedik[2] ·
Hakan Ferhatosmanoğlu[2]

Published online: 27 January 2017

Abstract We are witnessing an enormous growth in social networks as well as in the volume of data generated by them. An important portion of this data is in the form of graphs. In recent years, several graph processing and management systems emerged to handle large-scale graphs. The primary goal of these systems is to run graph algorithms and queries in an efficient and scalable manner. Unlike relational data, graphs are semi-structured in nature. Thus, storing and accessing graph data using secondary storage requires new solutions that can provide locality of access for graph processing workloads. In this work, we propose a scalable block formation and layout technique for graphs, which aims at reducing the I/O cost of disk-based graph processing algorithms. To achieve this, we designed a scalable MapReduce-style method called ICBL, which can divide the graph into a series of disk blocks that contain sub-graphs with high locality. Furthermore, ICBL can order the resulting blocks on disk to further reduce non-local accesses. We experimentally evaluated ICBL to showcase its scalability, layout quality, as well as the effectiveness of automatic parameter tuning for ICBL. We deployed the graph layouts generated by ICBL on the Neo4j open source graph database, http://www.neo4j.org/ (2015) graph database management system. Our results show that the layout generated by ICBL reduces the query running times over Neo4j more than 2× compared to the default layout.

✉ Abdurrahman Yaşar
ayasar@gatech.edu

Buğra Gedik
bgedik@cs.bilkent.edu.tr

Hakan Ferhatosmanoğlu
hakan@cs.bilkent.edu.tr

[1] College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA

[2] Department of Computer Engineering, Bilkent University, Bilkent, 06800 Ankara, Turkey

Keywords Graph management systems · Locality · Layout · Large scale graphs · Database management · Distributed systems

1 Introduction

We are witnessing an enormous growth in social networks and the volume of data generated by them. An important portion of this data is in the form of graphs, which are popular data structures used to represent relationships between entities. For instance, the graph structure may represent the relationships in a social network, where finding communities in the graph [9] can facilitate targeted advertising. In the telecommunications (telcos) domain, call details reports (CDRs) can be used to capture the call relationships between people [25], and locating closely connected groups of people can be used for generating promotions.

With the rise in the availability and volume of graph data, several graph processing and management systems have been introduced to handle large-scale graphs [12,15,18,21,23,24,31,39]. The primary goal of these systems is to manage large graphs and execute graph algorithms on them in an efficient and scalable manner. In this work, we focus on disk-based graph management systems [15,26], and propose the first parallel and scalable MapReduce (MR) based block formation and layout technique for graphs. Unlike relational data, graphs are semi-structured in nature. Thus, storing and accessing graph data using secondary storage requires new solutions that can provide locality of access for graph processing workloads.

Many graph algorithms rely on the fundamental operation of *traversal* and exhibit high access locality [34]. Given that a vertex is visited during a traversal, it is quite likely that the neighbors of this vertex will be visited shortly after. For instance, an n-hop breadth first search around a vertex exhibits high locality. This observation has motivated block-based disk layouts where the neighbor lists of vertices that are highly connected (e.g., form a community) are placed into the same disk block [14]. This reduces the number of blocks read, which reduces I/O. It also avoids the costly disk seeks, since chasing blocks requires seeking to different areas of the disk.

In this paper, we propose a novel distributed block formation and layout technique for large-scale graphs, which aims at reducing the I/O cost of disk-based graph processing algorithms. To achieve this goal, we designed a scalable MR style method called ICBL, which can divide the graph into a series of disk blocks that contain subgraphs with high locality, as well as order these blocks on disk to create a layout that reduces non-local accesses. In this paper, we describe the ICBL method, including the challenges that arose in applying ICBL in practice, the solutions applied, and an experimental evaluation showcasing its effectiveness.

Identifying vertices that are ‘close’ with respect to locality of access during execution of graph algorithms is a challenging problem. Although neighbor lists of vertices and their similarity give some information about locality, it is not sufficient. To illustrate, we can think two hop neighbors of a vertex. Although the neighbor lists of these vertices may have very few common neighbors, in a large graph we can certainly define them as close vertices. Accordingly, there should be a diffusion factor for each vertex, which can vary based on the graph size. In this work, we use random walks to produce *diffusion sets* of vertices. The idea behind building diffusion sets is simple: for each vertex, do some number of random walks and assign weights to vertices visited during the random walks. The resulting weighted sets of vertices can be used to define closeness between the originating vertices. At this point, we run into another challenge, namely defining the number of random walks and their lengths, based on the graph characteristics. We address this challenge by automatically tuning ICBL parameters.

Once the closeness between vertices is defined, we can use it to form disk blocks by co-locating close vertices within the same blocks. This could be achieved by using bottom-up methods from the literature, such as hierarchical clustering. Yet, these methods have high computational complexity, leading to prohibitive costs for large-scale graphs. Thus, forming the disk blocks in a scalable manner is a challenging problem. In this work, we use a coarse partitioning algorithm to divide the large graph to in-memory processable sub-graphs. This coarse partitioning gives us the ability to apply a computationally heavier block formation algorithm on these sub-graphs, in parallel.

Since the size of the disk blocks are relatively small compared to the graph size, the generated blocks are expected to contain many connections to other blocks. Therefore, to better benefit from locality of access, they need to be ordered on disk by taking into account the inter block connections. In this work, we solve the problem of graph block ranking using a label-based layout algorithm that is piggybacked on block formation. The layout algorithm orders the blocks based on their labels, which are generated as part of the block formation stage, roughly capturing the position of the blocks within the similarity based hierarchical merge process.

In the literature, block formation and layout for graphs has been considered [14], yet the solutions are not parallel or scalable. When considering the size of social media graphs and big data workloads, performing block formation and layout in a scalable manner becomes an important task. In this work, we achieve scalability by implementing all parts of our proposed ICBL solution as MR jobs. While our solution is tailored towards disk-based graph management systems that rely on block-based organization of data, it can also be applied to recent vertex programming systems with block/sub-graph based parallel processing [33,35,39].

In summary, we make the following contributions:

- We propose a block formation and layout technique called ICBL for large-scale graphs. ICBL is aimed at increasing the performance of disk-based graph management systems by improving the access locality of I/O.
- We develop MR-based algorithms to implement ICBL, making the process scalable, so that large-scale graphs can be divided into disk blocks and laid out on the disk using distributed processing.
- We propose evaluation metrics for measuring the efficacy of the ICBL technique and present an experimental evaluation showcasing its disk layout quality and running time scalability.

- We deploy the graph layouts generated by ICBL on the Neo4j [26] graph database management system to understand the impact of the layouts generated by ICBL on the performance of query evaluation.

Our experimental results show that the layout generated by ICBL reduces the query running times over Neo4j by more than 2× compared to the default layout.

2 Problem definition

Most graph analytics require graph traversals, where vertex access patterns follow the connectivity structure of the graph. If the graph is laid out on the disk without considering these patterns, the traversal operations may cause too many I/O operations. This can create a bottleneck for graph processing and management systems. Therefore, storing and accessing graph data using secondary storage requires new solutions that can provide locality of access for graph processing workloads.

Locality of access for graph analytics executing on disk-based graph processing systems can be increased by locating graph vertices that are ‘close’ with respect to connectivity structure of the graph close on the disk as well. Figure 1 illustrates this. In the figure, we have a graph with 18 vertices stored on 6 blocks. Storing vertices in blocks aims to put close vertices together and increasing the locality of access. However, after generating locality-aware blocks, we still need to order these blocks on disk because of the inter-block edges. In summary, our problem is composed by two sub-problems: (i) locality-aware block generation, and (ii) ranking and ordering these block on disk.

Illustrative example assume that as part of a graph analytics task we need to access all vertices that are within 2-Hop distance of vertex 0. 2-Hop neighborhood of vertex 0 contains four vertices: 1, 2, 3, and 7. In the first scenario, we consider that the assignment of vertices to blocks is being done randomly. In this case, the four vertices could have been assigned to different blocks, which would result in 4 block accesses with a total of 12 vertex reads, resulting in 42% success rate (number of vertices used per vertex read). However, if we consider the block structure that is given in Fig. 1, we end up with two block accesses with a total of four vertex reads, resulting in 83% percent success rate. As we can see in this example, locality-aware block generation decreases the number of block accesses and increases I/O efficiency.

Locality-aware block generation is highly critical in decreasing the number of reads from disk, and ultimately, in optimizing the efficiency of the graph database system. However, if our secondary storage is a hard disk, seek time becomes important as well. In our running example, we need to access a number of blocks and if these blocks are randomly scattered on the disk, then to read a relatively small number of blocks, we would spend too much seek time. For instance, let us assume that blocks are ordered randomly on the disk as follows: 5, 2, 3, 4, 0, and 1. We need to access all vertices that are in 2-hop distance from vertex 0. To start, we need to access block 0, which is in the fifth position. Later, we must access block 2, which is in the second position. This means that the disk needs to first seek to position 5 and then seek around back to position 2. However, if we use the layout that we defined in Fig. 1, that is 0, 2, 1, 4, 3, and 5, we would avoid the additional seek. Since blocks 0 and 2 are sequential,

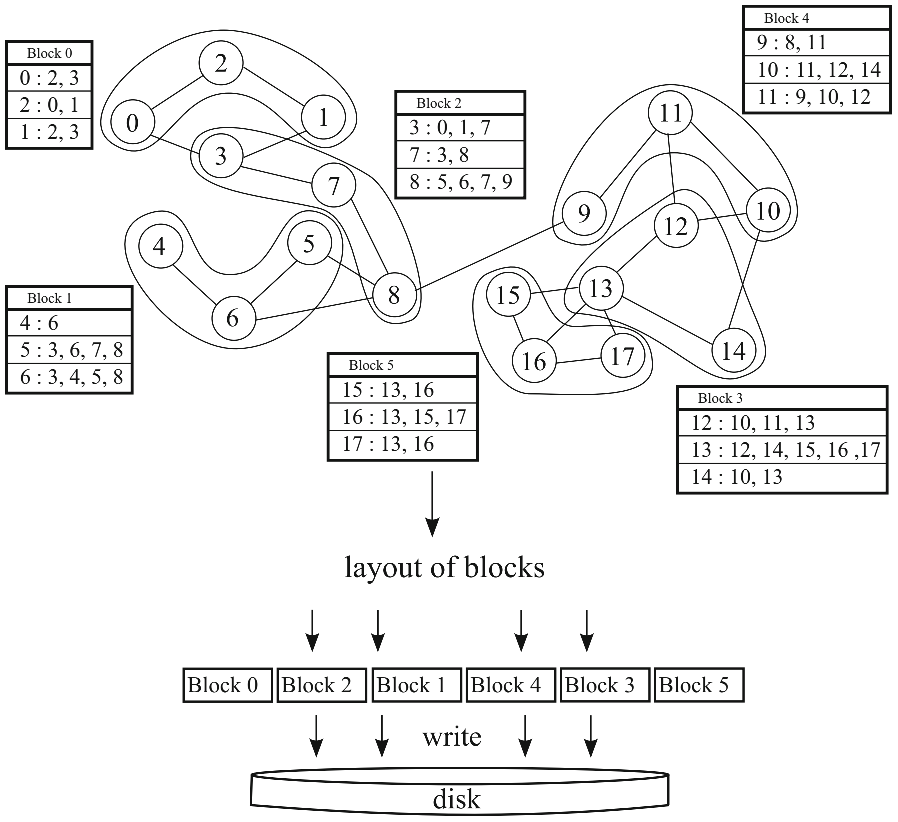

Fig. 1 Toy graph illustrating block formation and ranking

accessing these two blocks requires only a single seek. In conclusion, with a smart ordering seek time can be decreased to improve I/O efficiency.

2.1 Notation

An undirected graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. An edge is denoted as $e = (u, v) = (v, u) \in E$, where $u \neq v$ and $u, v \in V$. The neighbor list of a vertex $u \in V$ is denoted as N_u, and defined as $N_u = \{v \in E \mid (u, v) \in E\}$. $\mathcal{N}$ represents the set of all neighbor lists, that is $\mathcal{N} = \{N_v \mid v \in V\}$. For instance, if we consider Fig. 1, the neighbor list of vertex 0 is $N_0 = \{2, 3\}$ and $\mathcal{N}$ is $\{N_0, N_1, \ldots, N_6\}$.

Given a graph, we generate a set of blocks, denoted by $\mathcal{B}$. Each block $B \in \mathcal{B}$ contains at least one vertex and its neighbor list. Thus we can view a block as a non-empty subset of the set of all vertex–neighbor list pairs. Formally, $\forall B \in \mathcal{B}, B \subset \{(u, N_u) \mid u \in V\}$ and $|B| > 0$. Blocks do not share their elements, that is $\forall_{\{B, B'\} \subset \mathcal{B}}, B \cap B' = \emptyset$. We denote the set of vertices in a block B as $V_B = \{u \mid (u, N_u) \in B\}$ and the set of neighbor lists as $\mathcal{N}_B = \{N_u \mid (u, N_u) \in B\}$. The set of blocks cover the entire graph G, that is $V = \bigcup_{B \in \mathcal{B}} V_B$. Finally, each block is limited in size by a block size threshold denoted by S. Let $s\colon \mathcal{B} \rightarrow \mathbb{N}$ be a function that assigns a size to a block, then we have $\forall B \in \mathcal{B},\ s(B) \leq S$.

We assume that blocks are laid out on the disk sequentially. The place of a block B on the disk is determined by its rank, denoted by $r(B)$. The rank of a block is simply the number of blocks that have been written before it. We have $0 \leq r(B) < |\mathcal{B}|$, and $\forall_{\{B,\ B'\} \subset \mathcal{B}},\ r(B) \neq r(B')$. Similarly, rank of a vertex u, $r(u)$; is equal to the rank of the block B where $u \in B$. Finally, we define a function $d\colon \mathcal{B} \times \mathcal{B} \rightarrow \mathbb{N}$ that represents the distance between two blocks on the disk. We have $d(B,\ B') = |r(B) - r(B')|$.

2.2 Problem formulation

Our problem has two aspects, namely *block formation* and *block ranking*. In the block formation problem, the aim is to generate blocks with high locality. We define the locality of a block B using a metric that measures how well connected the vertices within the block are and how well separated they are from the vertices in other blocks, denoted by $L(B)$. Thus, the goal is to maximize the total locality over all blocks, denoted by $L = \sum_{B \in \mathcal{B}} L(B)$.

In the block ranking problem, the aim is to assign close ranks to blocks that have many edges connecting them, so that they are close on the disk. We define the ranking locality of a block B using a metric that measures the on-disk distance of B to other blocks it has edges into, denoted by $R(B)$. Thus, the goal is to maximize the total locality over all blocks, denoted by $R = \sum_{B \in \mathcal{B}} R(B)$.

2.3 Locality measures

Evaluation of our proposed system depends on the definition of block and block ranking localities. We now formally define these localities.

2.3.1 Block locality

Locality of a block can be defined using two concepts: conductance and cohesiveness. Conductance is commonly used for graph partitioning. In our context it is defined as the ratio of the number of edge cuts to the total number of edges in a block. An edge $\{u,\ v\}$ of a block B is considered as an edge cut if the destination vertex is not contained within block B. Formal definition of conductance is as follows:

$$C^d(B) = \frac{|\{(u,\ v) \in E \mid |\{u,\ v\} \cap V_B| = 1\}|}{|\{(u,\ v) \in E \mid |\{u,\ v\} \cap V_B| > 0\}|}. \tag{1}$$

For example conductance of Block 0 in Fig. 1 is $C^d(B_0) = 2/4 = 0.5$. Because, out of the four edges in the block, two are going out: (0, 3) and (1, 3).

Conductance of a block is not sufficient to determine the locality of a block. What is missing is the cohesiveness of the block. Cohesiveness is generally used for finding highly connected regions or communities in graphs. In this work we define cohesiveness of a block as the number of vertex pairs that are connected to each other via an edge in the block, divided by the total number of vertex pairs. Denoted by C^h, cohesiveness is formally defined as follows:

$$C^h(B) = \frac{|\{(u, v) \in E \mid u, v \in V_B\}|}{|B| \cdot (|B| - 1)/2}. \tag{2}$$

Again, if we consider Block 0 in Fig. 1, cohesiveness of the block becomes $C^h(B_0) = 2/3 = 0.66$. Because in block there are two connected pairs of vertices, out of three possible connections.

These two metrics are complementary. Impact of dangling edges is captured by conductance and connectivity within a block is captured by cohesiveness. To obtain a high locality block, we need to increase cohesiveness, while decreasing conductance.

As a result, we define the locality of a block B, denoted by $L(B)$, as the geometric mean of cohesiveness and one minus the conductance. That is:

$$L(B) = \sqrt{C^h(B) \times \left(1 - C^d(B)\right)}. \tag{3}$$

Finally, if we apply this formula to Block 0, we obtain:

$$L\,(B_0) = \sqrt{0.33 \times (1 - 0.5)} = 0.41.$$

.

2.3.2 Ranking locality

We define ranking locality in terms of the distance between blocks of neighboring vertices. Let us denote the ranking distance a vertex $u \in V$ has to its neighbor vertices by $R(u)$. Formally, we have:

$$R(u) = \sum_{v \in N_u} d(r(u), r(v)). \tag{4}$$

Then the ranking locality for a block B is defined as:

$$R(B) = 1 - \frac{\sum_{u \in V_B} R(u)}{d_{max} \times \sum_{u \in V_B} |N_u|}. \tag{5}$$

Illustrative example assume that as part of our evaluation we are computing ranking locality of the block B_0, which is given in Fig. 1. This block has vertices 0, 1 and 2. For instance $R(0) = d(r(0), r(2)) + d(r(0), r(3)) = 0 + 1 = 1$. Using the same formula, we obtain $R(1) = 1$ and $R(2) = 0$. After computing rankings of all vertices in that block, we can compute the ranking of the block B_0 as follows: $R(B_0) = 1 - \frac{1+1+0}{5*(2+2+2)} = 1 - \frac{2}{30} = 0.93$. In this example, the maximum distance between blocks is $d_{max} = 5$.

In this formula, d_{max} represents the maximum possible distance in the layout. We have: $d_{max} = max_{u,v \in V} d(r(v), r(u))$. When there are no edges going outside of a block, the ranking locality is 1. This is the ideal scenario. The ranking locality is in the range [0, 1].

3 Solution overview

In this section, we give an overview of our solution to scalable layout of large-scale graphs. Our approach, named ICBL,[1] consists of a multi-stage process, where each stage can be implemented in a scalable manner using MR style parallelism.

3.1 General approach

ICBL has four major stages. The first stage identifies the diffusion sets of vertices. The second stage performs coarse partitioning of the graph based on locality. It uses the diffusion sets from the first stage to guide the partitioning. The last two stages are used to form blocks and rank them. The forming of blocks and their ranking are implemented in an integrated manner to reduce the overhead of having an extra stage in the MR flow. Figure 2 illustrates these stages.

3.1.1 Identifying diffusion sets

Diffusion set of a vertex is a summarized representation of its neighborhood in the graph, not limited to single-hop neighbors. It can be used to define closeness between vertices. To identify the diffusion set of a vertex, we perform random walks starting from the vertex and record the vertices visited, together with the number of times they have been visited, during the random walks. The end result is a weighted set of vertices. We perform t random walks, each of length l. If we choose small values for l and t, then the neighborhoods will be sparse and thus similarities among neighborhoods of close vertices will be low. Conversely, if we choose large values for l and t, then many neighborhoods will end up looking similar, even if the vertices are not close. Also, large values will increase the computation time significantly, as diffusion sets are computed for each vertex. We address tuning of l and t in Sect. 4.1.

3.1.2 Coarse partitioning

After identifying diffusion sets for each vertex in the graph, we divide the graph into k vertex-disjoint sub-graphs. Vertices that are close based on the similarity of their diffusion sets are co-located on the same sub-graphs, as much as possible. The goal of the coarse partitioning is to create sub-graphs that can fit into the memory available on a single machine. Furthermore, coarse partitioning also helps us create sufficiently small sub-graphs that are suitable for executing computationally more expensive block

[1] Acronym is formed by the initial letters of the four solution stages.

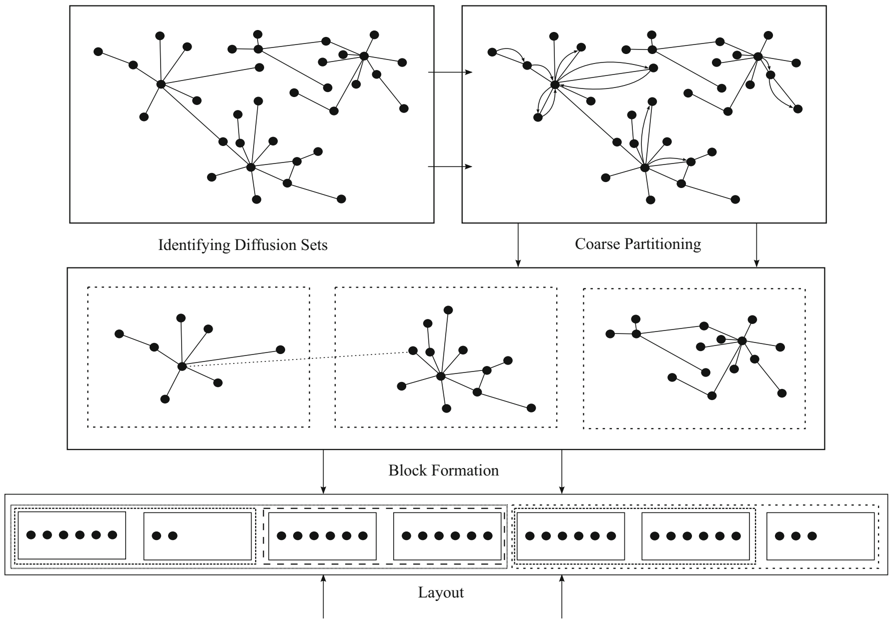

Fig. 2 Solution overview

formation algorithms inspired by hierarchical clustering. Naturally, as the input graph becomes larger in size, the number of partitions we need to create, that is k, increases as well. We address the tuning of k in Sect. 4.2.

3.1.3 Block formation

Block formation is performed in a bottom-up fashion. Initially, each vertex is in a partition by itself. Then we successively merge pairs of partitions to create bigger partitions. Among possible pairs, we pick the one that minimizes the distance between the diffusion sets of the vertices in the partitions. We further detail this in Sect. 4.3. If a partition exceeds the block size threshold, a block is formed. The block formation completes when all vertices are assigned to a block.

3.1.4 Layout

Layout is performed in an integrated manner as part of block formation. When the block formation algorithm finalizes a block, the layout algorithm assigns a *rank label* to the block. This rank label is a multi-segment string that approximates the location of the block within the hierarchical merge-tree of the vertices. Ordering the resulting blocks based on their rank labels gives their rank. The base layout algorithm only orders blocks within the same coarse partition, as the layout is performed independently for different partitions. A post-layout algorithm applies the same logic to order the coarse partitions, to achieve the final global ranking.

3.2 Scalability

Since our aim is to perform locality-aware block formation and layout for large-scale graphs, scalability is a primary concern. Therefore, ICBL is designed to be run as a series of MR tasks.

First, an MR task transforms the input graph given in the form of an edge list into an adjacency list formatted graph. This step is not needed if the input graph is already in the adjacency list format.

Second, we use two MR tasks to form the diffusion sets. The first task is responsible for performing l random walks and forming the vertex visit lists. The second job uses these lists to assign weights to vertices and form the final diffusion sets.

Third, we run a series of MR tasks to perform the coarse partitioning. The coarse partitioning is implemented as a variation of iterative k-means clustering. A first MR task is used to form initial partition centroids and the remaining tasks are used to perform a single iteration of a k-means clustering algorithm.

Last, we use an MR task to run the block formation and layout for each one of the coarse partitions we have created in the earlier stage.

4 Scalable block formation and layout

In this section, we discuss the details of the four stages comprising ICBL. For each stage, we describe parameter tuning and scalable implementation techniques.

4.1 Identifying diffusion sets

Diffusion set of a vertex v, denoted by $\mathcal{D}_v$, is used to capture the close vertices around v based on the vertices visited during random walks that start from v. To find $\mathcal{D}_v$, we apply t random walks around v, each of length l. We compute the diffusion sets for all vertices in the graph and implement it in a scalable manner using MR. An important and challenging aspect of identifying diffusion sets is tuning the parameters k and l based on the graph size and structure, which we discuss next.

4.1.1 Choosing t

Number of random walks (t) is critically important because if we set a too small t value, then the diffusion sets of vertices become very sparse and defining similarity of vertices using these sets becomes ineffective. Otherwise, if we set a too large t value, then the computation cost significantly increases without any benefit in terms of creating a diffusion set that can capture vertex similarity.

For a given graph, we define f as a cumulative distribution function of degrees, such that for $x \in \mathbb{N}$ $f(x) = P(d \leq x)$. In other words, $f(x)$ is the fraction of vertices that have a degree d less than or equal to x. Then we choose t as follows:

$$t = min\{x \colon f'(x) \leq \epsilon\}. \tag{6}$$

Here, f' is the function which gives the slope between x and $x+1$. In effect, we pick the smallest degree for which the distribution function's slope reaches ϵ. Our experimental evaluation has shown that choosing $\epsilon = 1$ gives robust results for varying graph sizes.

4.1.2 Choosing l

Vertex similarities are directly related to the setting of l. With large l values, the number of unique vertices that appear in diffusion sets increase and all vertices become similar. On the other hand, with small l values, the effectiveness of diffusion sets decreases as they become dissimilar even for close vertices.

In order to decide l, the first thing we should know is the diameter of the graph. Since social graphs exhibit small world phenomenon, their diameter can be estimated as the natural logarithm of the number of vertices they have, that is $ln(|V|)$ [36]. Accordingly, l should be at most $ln(|V|)$. Recall that after finding diffusion sets, we apply a coarse partitioning algorithm to divide the graph into k sub-graphs. Therefore, we choose l so as to cover the space within a sub-graph, as follows:

$$l = 1 + \lceil ln(|V|)/k \rceil. \tag{7}$$

4.1.3 MR implementation

t–l random walks are implemented via l repeated MR jobs, each one producing the vertices visited during the next hop of the random walks, followed by a final MR job for creating the diffusion sets. During the first iteration, the mapper takes the entire graph as input in the form of a series of vertex–neighbor list mappings. For each vertex, it chooses t random nodes from the neighbor list and sends each vertex, neighbor pair to the reducer. The reducer is an identity reducer in the first iteration. The result is a file that contains the *initiator* vertex as the key, and the *visited* vertex as the value. After the first iteration, $l-1$ identical MR jobs are run. In these iterations, the mapper takes the original graph and the output from the previous step as input. If a key/value pair comes from the original graph, then the mapper sends this pair directly to the reducer. If not, it switches the initiator with the visited and sends the resulting pair to the reducer. This swapping enables joining the visited vertex with its neighbor list, so that the next vertex to visit can be determined at the reducer side. For each visited vertex, the reducer collects the initiator vertices plus the neighbor list of the visited vertex. For each initiator, it determines the next visited vertex using the neighbor list of the current one, and outputs an initiator, next visited vertex pair. Algorithms 1 and 2 give the pseudo-codes for the mapper and the reducer for the iterative steps of the random walks, respectively.

When l iterations are completed, the final MR job combines all intermediate files and outputs the diffusion sets. Assigning weights to vertices in the diffusion sets is an important step performed by this last task, because it identifies the vertices that are commonly visited (closer). We tested our system with several alternatives for the weight assignment:

Algorithm 1: Random Walk Mapper

```
Param  : $t$, the number of random walks; $isFirst$, whether this is the first job
Input  : $\langle key, value\rangle$
if $isFirst$ then
    let $\langle v, N_v\rangle = \langle key, value\rangle$
    for $t$ times do
        $u \leftarrow N_v[rand()]$
        output $\langle v, u\rangle$
else
    if $value$ is a neighbor list then
        let $\langle u, N_u\rangle = \langle key, value\rangle$
        output $\langle u, N_u\rangle$
    else
        let $\langle v, u\rangle = \langle key, value\rangle$
        output $\langle u, v\rangle$
```

Algorithm 2: Random Walk Reducer

```
Param  : $isFirst$, whether this is the first job
Input  : $\langle key, values\rangle$
$N \leftarrow nil$                       ▷ neighbor list of last visited vertex
$V \leftarrow []$                        ▷ initiator vertices for last visited vertex
if $isFirst$ then
    let $\langle v, U\rangle = \langle key, values\rangle$
    foreach $u \in U$ do
        output $\langle v, u\rangle$
else
    let $u = key$
    foreach $value \in values$ do
        if $value$ is a neighbor list then
            let $N_u = value$
            $N \leftarrow N_u$
        else
            let $v = value$
            $V \leftarrow V + [v]$
    foreach $v \in V$ do
        output $\langle v, N[rand()]\rangle$
```

– non-weighted diffusion paths,
– occurrence count based weighted diffusion sets, and
– tf-idf based weighted diffusion sets.

Tf-idf based weights are computed by treating each diffusion set as a document and using the traditional term frequency times inverse document frequency formulation from Information Retrieval [30]. In our context, the term frequency is the weight of a vertex in the diffusion set. The inverse document frequency for a vertex is the logarithm of the ratio of the total number of vertices to the number of diffusion sets that contain the vertex.

4.2 Coarse partitioning

After identifying diffusion sets for each vertex in the graph, we divide the graph into k vertex-disjoint sub-graphs as part of the coarse partitioning stage. The goal of the coarse partitioning is to create sub-graphs that can fit into the memory available on a single machine. Furthermore, coarse partitioning also helps us create sufficiently small sub-graphs that are suitable for executing computationally more expensive block formation algorithms inspired by hierarchical clustering.

Our coarse partitioning algorithm is based on k-means [22]. As such, we first choose a set of k initial centers, denoted by $\mathcal{C}$, from the graph. Then, for each vertex $v \in V$, we find the closest center $c \in \mathcal{C}$ and assign v to the cluster of c. After all vertices are assigned, we obtain a list of vertices for each cluster, denoted as as V_c for center c. We then calculate the new centers, that is we update $\mathcal{C}$, by reducing V_c into a new center value replacing the old one. The process is repeated until convergence, detected based on comparing the difference between the new and old clusters to a threshold.

We now describe the various details of the algorithm, such as the distance metric we use, setting the value of k, and determining the initial centers. We then provide a brief description of the MR implementation.

4.2.1 Distance metric

To determine closeness of vertex pairs we need to define a distance metric. Since diffusion sets are just weighted sets of vertices, we use a weighted Jaccard distance for this purpose. *Jaccard similarity* of two sets S and T is the ratio of the size of their intersection to the size of their union, that is $\frac{|S \cap T|}{|S \cup T|}$. If we apply this in our context for two vertices $u, v \in V$, we get $JS(u, v) = \frac{|\mathcal{D}_u \cap \mathcal{D}_v|}{|\mathcal{D}_u \cup \mathcal{D}_v|}$. As we mentioned before, the vertices in diffusion paths could be weighted. The intuition behind weighted diffusion sets is to improve closeness of vertices. For example, if a vertex frequently appears in two diffusion sets, then we should compute a closer distance for these sets. In that case we have a weighted Jaccard similarity, defined as $JS_w(u, v) = \frac{\sum_{x \in \mathcal{D}_v \cap \mathcal{D}_u} min\{w(x, \mathcal{D}_v), w(x, \mathcal{D}_u)\}}{\sum_{x \in \mathcal{D}_v \cup \mathcal{D}_u} max\{w(x, \mathcal{D}_v), w(x, \mathcal{D}_u)\}}$. Here, $w(x, \mathcal{D})$ represents the weight of vertex x in diffusion set $\mathcal{D}$. After defining the similarity between two vertices, the Jaccard distance between them is simply: $JD(u, v) = 1 - JS_w(u, v)$.

4.2.2 Choosing k

Tuning the k parameter is crucial because coarse partitioning aims to divide the graph into in-memory processable sub-graphs for the following block formation stage. Therefore, if we choose a too small k value, then we can run out of memory in the block formation stage. On the other hand, if we choose a too large k value, then we increase the processing time for the coarse partitioning stage and we also lose the locality effect that is needed for the block formation stage to form blocks with high locality. Assume that all cores in our cluster has M byte of memory and a vertex with its diffusion set is s bytes. Then we choose k as follows:

```
Algorithm 3: Coarse Partitioning Mapper
Param : $\mathcal{C}$, set of centers, where for $c \in C$, $c.id$ is the center id and $c.S$ is the diffusion set for the
        center.
Input : $\langle key, value\rangle$
let $\langle v, \mathcal{D}_v\rangle = \langle key, value\rangle$
$c \leftarrow \text{argmin}_{c \in \mathcal{C}} JD(\mathcal{D}_v, c.S)$
output $\langle c.id, \mathcal{D}_v\rangle$
```

$$k = \lceil s \cdot |V| / \sqrt{0.8 \times M} \rceil. \tag{8}$$

In summary, we make k as small as possible without utilizing more than 80% of the main memory available to a core in the system.

4.2.3 Initial centers

One way of creating the initial centers is to choose them randomly. However, in our experiments this has caused unstable performance, both in terms of convergence of the coarse partitioning stage as well as the locality of the resulting blocks. Instead, we came up with a more effective way of setting the initial centers. The idea is to pick k vertices that are distant to each other and have high degrees. These can be considered as influence centers in the graph. To compute them, we added an MR job to the system to sort the vertices by degree. We then process this list, starting from the highest degree vertex. If a vertex has a distance 0.9 or more to all of the previously selected ones, we select it as a center vertex. We stop when k vertices are selected.

4.2.4 Deciding center size

Cluster centers are weighted sets, just like the diffusion sets. Recall that at the end of each iteration of k-means, we have to form new centers. The size of these centers is also an important factor. If we choose a too small size, then coarse partitioning converges too fast and the resulting clustering has poor locality. If the size is too large, then this delays convergence. We set the center size to the average length of the diffusion paths within a cluster. In our empirical study, this setting has resulted in good quality sub-graphs and has shown good convergence behavior.

4.2.5 MR implementation

Coarse partitioning is implemented via repeated sequential MR jobs. The first iteration takes a set of initial centers denoted by $\mathcal{C}$. Remaining iterations produce the new centers for their following iterations until the final MR job, which produces the final clustering. We produce new centers by counting the number of occurrences of vertices in each cluster and keeping the most frequent ones. Algorithms 3 and 4 give the pseudo-codes for the mapper and the reducer for the coarse partitioning stage, respectively.

Algorithm 4: Coarse Partitioning Reducer

```
Param : isLast, whether this is the last job
Input : $\langle key,\ values \rangle$
$O \leftarrow \{\}$                                   ▷ Map from vertex to in-cluster occurrence count
$size \leftarrow 0$                                   ▷ Average diffusion set size in cluster
let $cId = key$                                       ▷ key is the cluster id
if not isLast then
    foreach $value \in values$ do
        let $\mathcal{D} = value$                     ▷ each value is a diffusion set
        foreach $v \in \mathcal{D}$ do
            $O[v] \leftarrow O[v] + 1$
        $size \leftarrow size + |\mathcal{D}|$
    $size \leftarrow size/|values|$
    $\mathcal{D} \leftarrow$ argtop-$k_{v \in O}\, O[v]$, where $k = size$
    $c \leftarrow tuple(id{=}cId,\ S{=}\mathcal{D})$
    output $\langle cId,\ c \rangle$
else
    foreach $value \in values$ do
        let $\mathcal{D} = value$
        $c \leftarrow tuple(id{=}cId,\ S{=}\mathcal{D})$
        output $\langle cId,\ c \rangle$
```

4.3 Block formation

During block formation, vertices are placed into partitions in a bottom-up fashion. Each vertex starts in its own partition and partitions are successively merged by picking the closest pair of partitions at each step. We define the closeness of two partitions as the minimum Jaccard distance between the diffusion sets of the vertices contained within. For partitions P and P', this is given as $\min\{JD(\mathcal{D}_u, \mathcal{D}_v) \colon u \in P \wedge v \in P'\}$. When the size of a potential block that would be formed by vertices in the partition without a block assigned so far exceeds the maximum block size, then a full block is formed and output. The block formation completes when all vertices are assigned to a block.

4.3.1 Super blocks

In large graphs that exhibit power law [27] degree distribution, popular nodes require special treatment. If we take the Twitter graph as an example, a user with millions of followers becomes an exceptional case because the size of his/her neighbor list exceeds the block size. In such exceptional cases, we divide the neighbor list of the vertex into multiple block sized segments. We refer to a block that points to multiple such segments as a *super block*.

4.3.2 Block labeling

We assign labels to blocks for helping with the last stage of the ICBL solution, that is layout. For this purpose, during the execution of the block formation algorithm, each partition maintains a label. This partition label is used to derive the block label later. It captures the merge history of partitions with respect to blocks. Initially, each partition

Algorithm 5: Block Formation Algorithm

```
Param : $S$, block size; $V$: set of vertices in the sub-graph
$\mathcal{B} \leftarrow \emptyset$    ▷ Blocks to be generated $\mathcal{P} \leftarrow \bigcup_{v \in V}\{tuple(l{=}str(v),\ i{=}false,\ V{=}[v], U{=}\{v\})\}$
while $|\mathcal{P}| > 1$ do
    $\{P,\ P'\} \leftarrow \text{argmin}_{\{P,\ P'\} \subseteq \mathcal{P}}$
            $\min\{JD(\mathcal{D}_u,\ \mathcal{D}_v) \colon u \in P.U \wedge v \in P'.U\}$
    ▷ Setup the partition label
    let $P_n = \text{argmin}_{P'' \in \{P,\ P'\}} |P''.U|$    ▷ Small partition
    let $P_x = P''\ s.t.\ P'' \neq P_n \wedge P'' \in \{P,\ P'\}$    ▷ Large part.
    if $P_n.i \wedge P_x.i$ then $P_x.l \leftarrow P_x.l + \text{“ :''} + P_n.l$
    else if $\neg P_x.i \wedge P_n.i$ then $P_x.l \leftarrow P_n.l$
    ▷ Merge the partitions
    $\mathcal{P} \leftarrow \mathcal{P} \setminus \{P_n\}$
    $P_x.U \leftarrow P_x.U \cup P_n.U$
    $P_x.V \leftarrow P_x.V \cup P_n.V$
    if $blockSize(P_x.V) \geq S$ then
        $P_x.i \leftarrow true$    ▷ Remember generation of block
        $k \leftarrow \max\{k' \colon blockSize(P_x.V[0{:}k]) \leq S\}$
        $V' \leftarrow P_x.V[0{:}k]$    ▷ Vertices to form a block
        $B \leftarrow \{(v,\ N_v) \colon v \in V'\}$    ▷ Form the block
        $\mathcal{B} \leftarrow \mathcal{B} \cup B$
        $P_x.V \leftarrow P_x.V \setminus V'$    ▷ Update unassigned vertices
return $\mathcal{B}$
```

has its vertex id as its label. When two partitions merge, this label is updated as follows: if the two partitions have not produced a block before, the new label is taken as the label of the larger partition. If only one of them has formed a block before, then its label is taken as the partition label. Finally, if both of the partitions have produced a block before, then the label is taken as the concatenation (using “ : ” as a delimiter) of the two labels, label of the bigger partition appearing on the left. When a block is produced, it gets the label of its partition, with an additional suffix (using “ . ” as a separator) representing the index among blocks generated with the same partition label. Figure 3 shows an example block formation process, where numbers represent the order in which the partitions are merged. The partition labels are indicated on tree edges representing the merges. Blocks are marked with dotted boxes and their block labels are indicated next to the boxes.

4.3.3 MR implementation

Block formation is implemented with a single MR job, making use of only the map operation. Each map performs block formation on one of the sub-graphs generated by the coarse partitioning stage and produces blocks with their associated labels. Algorithm 5 gives the pseudo-code for this process.

4.4 Layout

Social graphs exhibit small-world behavior, and thus most vertices are reachable from each other via a small number of hops. Therefore, even with locality-aware block

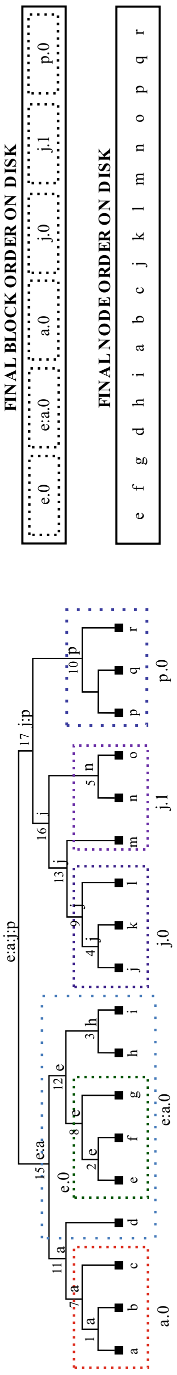

Fig. 3 Illustration of block layout

formation, we will have many edges crossing between blocks. With the layout algorithm, we aim to provide a locality-aware disk ordering for graphs by considering inter block similarities. Primary goal of the layout process is to store similar blocks close on disk.

The layout algorithm simply orders the blocks based on their labels that were generated as part of the block formation phase. Before performing a sort, we replace the vertex names that appear in the block labels with their order in the leaves of the hierarchical merge tree. Then sorting the blocks by their labels locate blocks that were close in the merge tree, close on the disk as well.

For instance, in Fig. 3, first nodes a and b are merged. then e and f, and so on. As you can see, we construct a tree in a bottom up manner. In this toy example, for brevity and ease of exposition, we assume that all vertices have the same degree d and size limit for a block is $3 \times (d + 1)$, thus only three vertices fit in a block. We observe that in the seventh iteration, the vertices $[a, b, c]$ reach the size limit and block formation algorithm generates them as a block. This block is labeled as $a.0$ by taking the partition label at the time of block generation (a in this case), and the index among the blocks that are generated with that partition label (0 in this case). This procedure continues to create blocks out of vertices $[e, f, g]$, $[j, k, l]$, $[m, n, o]$, and $[p, q, r]$.

In Fig. 3, the block that contains vertices $[d, h, i]$ is different, because the vertices in this block are not contiguous at the leaf level. In the 11th step, the partition that contains d merges with the partition that has earlier produced block $a.0$. And in the 12th step, the partition that contains h and i merges with the partition that earlier produced block $e.0$. Finally, 15th step, we merge these two partitions. The resulting partitions gets the label $a{:}e$, because the constituent partitions both have produced blocks earlier. Since the number of vertices without assigned blocks in the partition reaches the maximum size, a new block that contains the vertices $[d, h, i]$ and has label $a{:}e.0$ is generated.

Finally, when block formation is completed, we order blocks by sorting their labels. The end result is seen at the bottom of Fig. 3.

Recall that this layout procedure is performed for each sub-graph, in parallel. Once the order of blocks with each sub-graph is determined, a sequential version of the same process is applied across sub-graphs, by treating each sub-graph as a virtual vertex and pre-computing the distances among them based on the number of edges going across. The end result is an ordering that specifies which sub-graph blocks go earlier on the disk.

5 Experimental evaluation

In this section, we evaluate our system with a focus on the impact of the proposed optimizations on the locality of the generated layout and the scalability of the block formation and layout process. Scalability experiments evaluate the running time of our ICBL algorithm as a function of the number of cores used and the size of the graph. Locality experiments evaluate the performance using locality metrics, as well as query running time using an industrial-strength graph database system.

Table 1 List of real-world graphs used in the experiments

Graphs	Number of vertices	Number of edges
ego-Facebook	4039	88,234
wiki-Vote	7115	103,689
wiki-Talk	2,394,385	5,021,410
com-Orkut	3,072,441	117,185,083
uk-2002	18,520,486	298,113,762
arabic-2005	22,744,080	639,999,458
uk-2005	39,459,925	936,364,282
twitter	41,700,000	1.47 Billion

5.1 Experimental setup

We first provide details on our implementation, evaluation environment, the datasets used, and the metrics employed in our evaluation.

5.1.1 Implementation

Our implementation was done in Java 1.7 using Hadoop v2.6. For evaluation of the coarse partitioning method we use Metis [16] graph partitioning tool and for evaluation of the layout we use Neo4j [26] graph database. For workload generation, we use R-MAT [6] implementation of Boost Library [32].

5.1.2 Environment

For running the ICBL algorithm, we used a cluster consisting of 8 machines with a total of 96 CPU cores. Each machine has two 6-core Intel Xeon E5-2620 2.00 GHz processors, 32 GB of memory, and 1 TB disk space from 4 IBM Server X 5400 SATA disks configured via RAID-5. The operating system used was CentOS GNU/Linux with the 2.6 kernel and ext4 file system.

5.1.3 Datasets

We use R-MAT [6] generated graphs, as well as real-world graphs obtained from SNAP [20], Kwak et al. [17] and WebGraph [4,5].

5.1.4 Synthetic data

In our experiments we use R-MAT generated power-law graphs with small world properties. The R-MAT graph generator provides an efficient way for generating large realistic graphs. The number of edges is taken as 20 times the number of vertices.

5.1.5 Real data

In addition to the R-MAT graphs, we also selected several small, medium, and large sized graphs from SNAP, listed in Table 1.

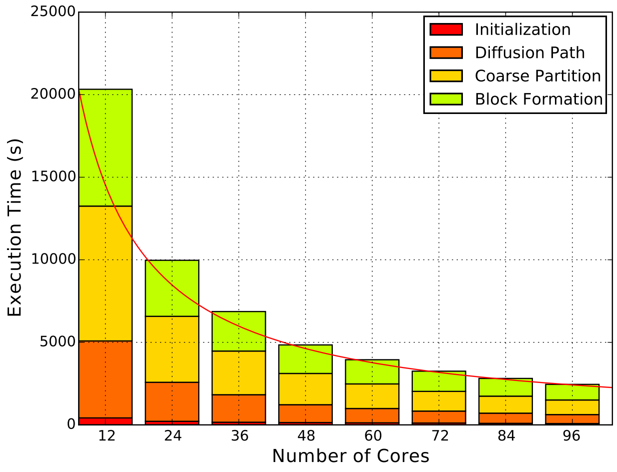

Fig. 4 Scalability w.r.t. number of cores

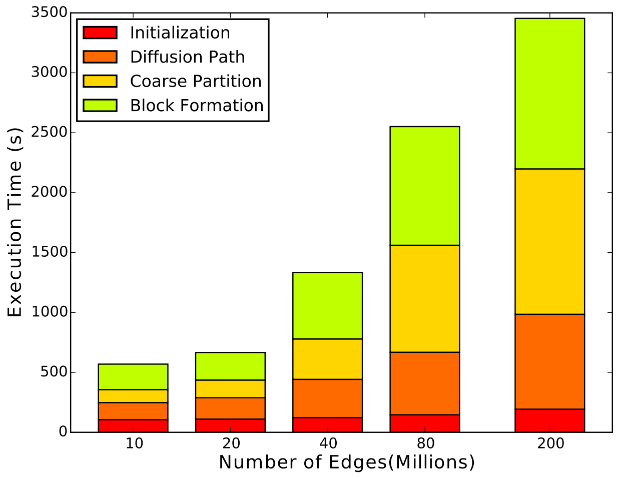

Fig. 5 Scalability w.r.t. number of edges

5.2 Scalability

Figure 4 shows the running time of the ICBL method as a function of the number of cores used. The graph used in this experiment is an 80 million edge R-MAT graph. Each bar represents the total amount of time the ICBL algorithm took to generate the disk layout. The different colored sub-bars represent the time taken by different stages on the ICBL method. The first sub-bar represents *initialization*, which is used to convert the initial graph from edge list representation to adjacency list representation. The second sub-bar represents forming the diffusion sets, and the third sub-bar represents coarse partitioning. The fourth and final sub-bar represents block formation, which also performs layout generation. The figure also shows an *ideal line* representing perfect scale-up. Figure 5 shows the running time with the same breakdown, but as a function of the number of edges.

We observe from Figs. 4 and 5 that initialization step takes negligible time compared to other stages, as it is very light on computation. Among the remaining stages, forming the diffusion sets is cheaper than coarse partitioning and block formation, but in general the distribution is quite balanced, especially with increasing number of cores. The most striking observation from Fig. 4 is about scalability. We see that ICBL's running time with increasing core sizes closely matches the running times represented by the ideal scale-up line.

We observe from Fig. 5 that the running time is sub-linear in the number of edges. In fact, running time is expected to be linear in the size of the diffusion sets, and not the number of edges. In particular, the *t* parameter is one of the key factors that determine the size of the diffusion sets. In our parameter selection policy, *t* does not increase proportional to number of edges of the graph, and instead it increases more slowly. This explains the sub-linear trend in Fig. 5.

5.3 Locality

In this section, we study the effectivenesses of our proposed optimizations on the locality of the layouts generated by ICBL.

5.3.1 Effectiveness of coarse partitioning

Coarse partitioning plays an important role in ICBL, as the localities of the generated blocks are affected by the quality of the sub-graphs generated by coarse partitioning. To understand the effectiveness of coarse partitioning, we compare it to a more traditional approach: graph partitioning.

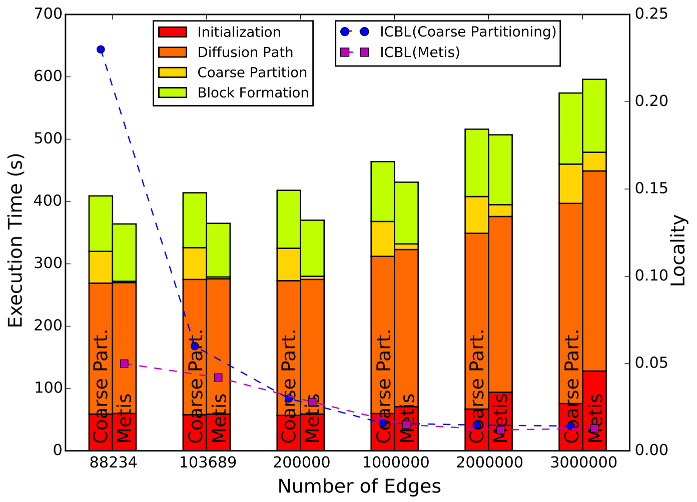

Fig. 6 ICBL with Metis and coarse partitioning

Metis [16] is one of the popular and effective graph partitioning methods in the literature and it produces high-quality graph partitions. Therefore, in this experiment, we compared the results from ICBL with those from a variant of ICBL where the coarse partitioning is replaced by graph partitioning. The graph partitioning aims to minimize the edge cut, while balancing the number of vertices in each partition. Figure 6 plots the locality of the resulting blocks, as a function of graph size. We use six different graphs for this purpose. The first two graphs are real graphs from SNAP, namely *ego-Facebook* and *wiki-Vote*, and the last four ones are generated using R-MAT.

From Fig. 6 we observe that for small graphs (especially the first real-world graph), ICBL with coarse partitioning can lead to improved locality compared to using ICBL with Metis. However, for larger graphs, the localities achieved by the two approaches are identical. We prefer coarse partitioning over Metis due to its scalability and integration into ICBL's Hadoop framework, as well as its good locality for large-graphs that is the focus of this work. Figure 6 also shows that Metis starts to take more time as the graph size is increased. Furthermore, pre-processing also starts to take more time for Metis, as the graph needs to be converted into the input format of Metis. The time taken by coarse partitioning, on the other hand, is not effected as much from the number of vertices, even though in absolute terms it takes more time than Metis for smaller graph sizes. For 300 million edges, ICBL with coarse partitioning starts to take less time compared to Metis. While parallel versions of Metis [19] can be used to bring the running time performance of Metis down and make it scale, we have not made this comparison, as the resulting locality is not going to be any better than for serial Metis.

5.3.2 Assigning weights

Having weighted diffusion sets helps us better capture similarity for vertices, which in turn is expected to improve block locality. To understand the impact of weight assignment on the locality of the generated blocks, we compared three alternatives schemes: non-weighted, occurrence counts as weights, and tf-idf weights computed over occurrence counts. For the weighted schemes, it is important to note that during random walks, the host vertex is assumed to be visited as the first vertex.

Figure 7 plots the execution time of ICBL (using the left y-axis) and locality (using the right y-axis), for different weighting schemes and for R-MAT generated graphs of different sizes (20, 40, 80, and 200 million edges).

We observe from Fig. 7 that for all graphs sizes, tf-idf based weight assignment improves locality compared to non-weighted and occurrence count based weighted cases, with relative improvements ranging from 20 to 50%. Since tf-idf based weights decrease the importance of very popular vertices in diffusion sets, this type of weight assignment improves the quality of sub-graphs that are generated with coarse partitioning by reducing the tendency of vertices to accumulate in one cluster.

5.3.3 Choosing centers

In this experiment we examine two center selection strategies, namely *random* and *distant*. The first selection strategy is to choose randomly selected k host vertices and their adjacency lists as centers. The second selection approach is to choose k most

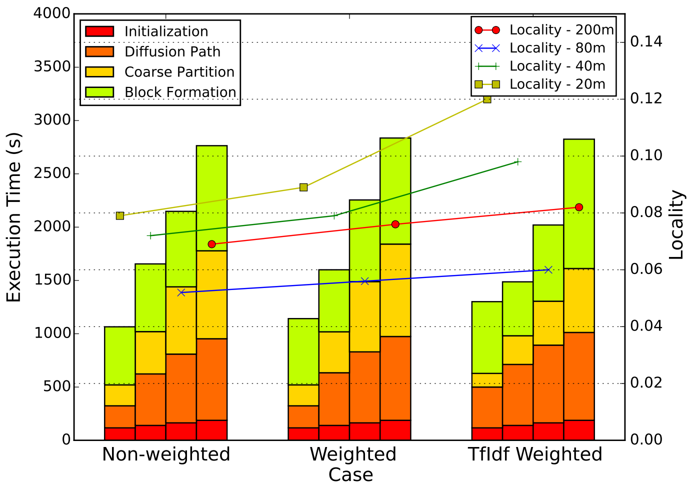

Fig. 7 Assigning weights to diffusion sets

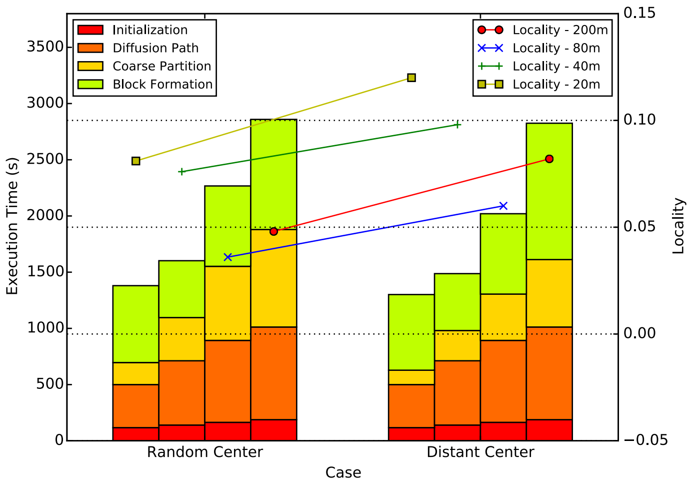

Fig. 8 Choosing initial centers

distant and highest degree host vertices and their adjacency lists as initial centers, as explained earlier in Sect. 4.2. For this experiment, we again used R-MAT-generated graphs.

Figure 8 plots the execution time of ICBL (using the left *y*-axis) and locality (using the right *y*-axis), for the two center selection schemes and for four different graph sizes (20, 40, 80, and 200 million edges).

We see that initial center selection strategies impact the convergence speed of coarse partitioning. Based on our experiments, we have observed that starting coarse partitioning with randomly selected centers from the graph sometimes requires more iterations to converge. The 40 million edge graph is a good example of this in Fig. 8, where the coarse partitioning takes almost two times longer with random center selection.

From Fig. 8, we also observe that initial center selection strategy impacts locality. For all graph sizes, the distant center selection strategy outperforms the random one, up to 30% in some cases.

Although distant center selection strategy improves locality and speeds up convergence, in some cases it also increases the time taken by the following stage of ICBL, that is block formation. This can be observed for the 200 million edge graph in Fig. 8. Still, ICBL with distant center selection completes faster than random selection, for all graph sizes. The reason block formation sometimes takes longer with distant center selection is that, higher quality sub-graphs formed by it may have higher skew in their sizes, resulting in load imbalance during the block formation stage.

5.3.4 Locality and length of diffusion paths

In this experiment we examine the effect of diffusion path length on locality. We apply ICBL with diffusion paths of length $l = 1$–3. We use R-MAT graphs for this experiment.

Figure 9 plots locality as a function of the diffusion path length (l), for graphs of different sizes. Each line shows locality for a different diffusion path length and the hexagons represent the locality for the length chosen by ICBL. We observe that, as the diffusion path length increases, the locality first increases and then decreases. The latter is because when we enlarge l too much, the diffusion set sizes increase and all sets become similar.

We observe that ICBL chooses the best diffusion path length for graphs with 20, 80 and 200 millions edges. For 40 millions edges, although ICBL couldn't choose the best length, we observe that there is a little locality difference with the best length. Therefore we can say that ICBL gains in terms of execution time by choosing a smaller l and loses little locality. For the graph which has 10 millions edges, ICBL chooses the worst length, because this is a small graph and can be handled with a smaller k. Since k and l are inversely proportional, ICBL chooses a larger l and performs poor on locality.

5.3.5 Locality and block size

In this experiment we examine the effect of block size on locality. We apply ICBL with blocks of size 32, 64, 128, 256, 512, and 1024 KBs. We use R-MAT graphs with differing sizes and measure locality.

Figure 10 plots locality as a function of the block size, for graphs of different sizes. The overall locality is shown on the left y-axis and $1 -$ conductance is shown on the right y-axis. Since cohesiveness has a term that graphs quadratically with the number of vertices in a block, it brings down the overall locality significantly. Thus, we also show conductance separately in this experiment. We observe that, as the block sizes increase, the conductance decreases. This is intuitive, as if there was only a single block, then conductance would have been 1. However, the overall locality decreases as the block size increases, due to the impact of cohesiveness.

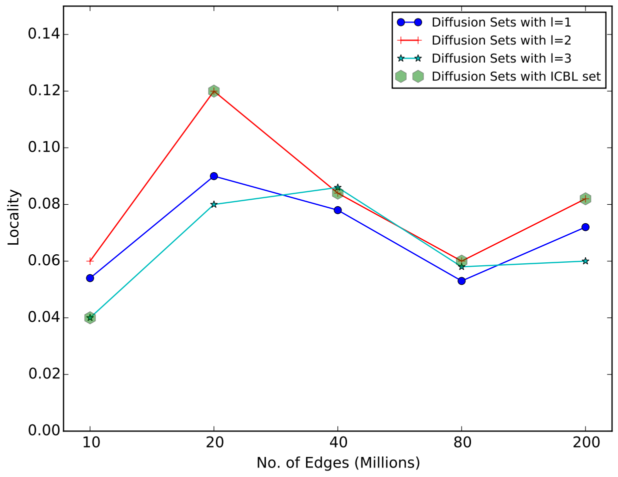

Fig. 9 Locality versus diffusion path length (l)

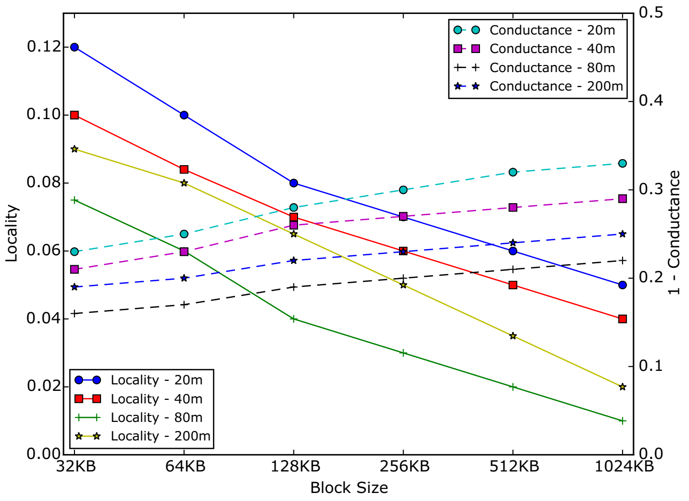

Fig. 10 Locality versus block size

5.3.6 ICBL and real graphs

In this experiment we examine the effectiveness of ICBL on real world graphs listed in Sect. 5.1.

Figure 11 plots locality and execution time for graphs of different sizes. The overall locality is shown on the right y-axis and execution time is shown on the left y-axis. We observe from Fig. 11 that initialization step takes negligible time compared to other stages, as R-MAT generated graphs. Among the remaining stages, performance is again similar to R-MAT generated graphs. Different than R-MAT graphs, for the real-world graphs, the locality is better for the larger sized graphs.

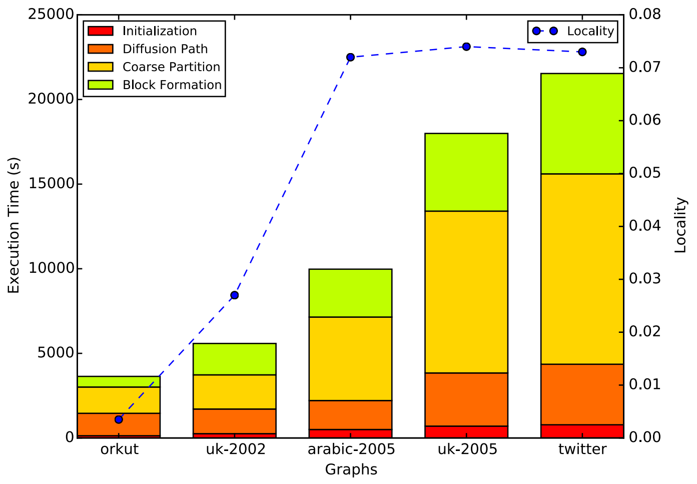

Fig. 11 Real graph experiments

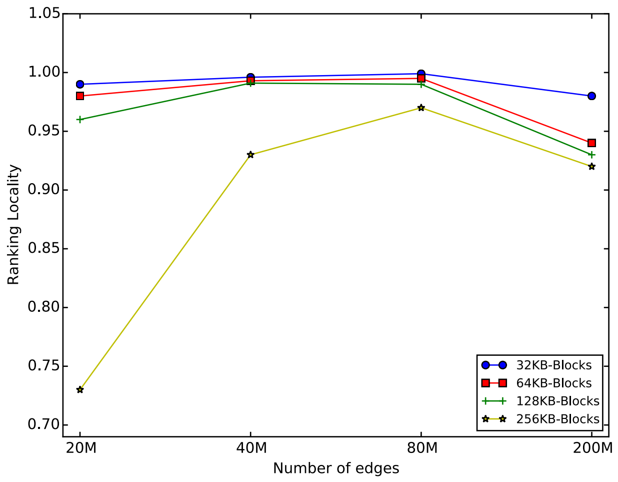

Fig. 12 Ranking locality versus graph size

5.3.7 Ranking locality

In this experiment, we evaluate ranking locality for different graph and block sizes. We use Eq. 2.3.2 to compute ranking localities over all disk blocks. We use distant center selection and tf-idf weight assignment strategies. The graphs used are R-MAT generated.

Figure 12 plots ranking locality as a function of graph size, for different block sizes. Overall, ranking localities are high. An important observation from the figure is about the sensitiveness of ranking locality to graph size. Small blocks are more resilient to changes in the graph size. In fact, 32 KB blocks have ranking localities almost

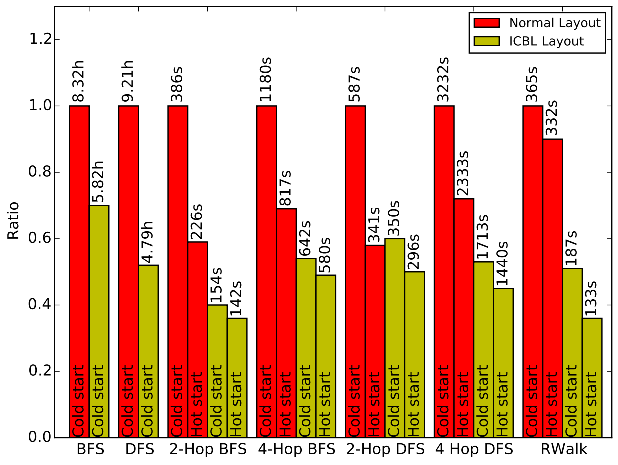

Fig. 13 Query running times with Neo4j

independent of graph size. On the other hand, 256 KB blocks show high variation in locality as the graph size changes, compared to smaller block sizes.

5.3.8 Query running times

To understand the impact of the layouts generated by ICBL on the performance of query evaluation in a graph database, we deployed the graph layouts generated by ICBL to the Neo4j [26] graph database management system. For this experiment, we used the 80 million edge R-MAT graph. To evaluate query performance, we used global BFS and DFS queries, limited hop BFS and DFS queries, and random walks. The limited hop queries were run 100 times and the average results are reported. These graph algorithms were implemented using the Java API provided by Neo4j [26].

Deployment of the ICBL generated layout to Neo4j is performed in two stages. First stage is for preparation and the second one is for generation of the Neo4j specific files on the disk. Neo4j stores graphs in separate files and uses a variation of edge list format to represent relationships between vertices. Since Neo4j doesn't have a clear notion of a block and uses edge lists, our adjacency list based block structure needs to be converted. In the preparation stage, we do this conversion in two steps. First we merge blocks according to layout order and obtain a single file, and second we transform this file into edge list format. After the edge list file is generated, we create a second file, which stores vertices in the order of their first appearance in the edge list. These two files become inputs of the second stage. In the generation stage, we create Neo4j specific files using MR jobs, consisting of join, union, and ascending sort.

Figure 13 shows the running times of the algorithms, normalized with respect to Neo4j's default layout (labeled as Normal in the graph). We also have absolute running times as annotations in the figure. We show running times for both cold start and hot start cases, except for the global queries for which a hot cache does not make a difference (since the query touches the entire database). We observe that the default layout of Neo4j has 43 and 92% higher running times compared to ICBL for the BFS and DFS algorithms, respectively. For the cold start case using limited hop queries, the default layout results in running times that are 1.5–2.5 times that of with ICBL. The relative results are similar even for the hot start case, except for 2-hop DFS where the normal layout and ICBL perform similarly.

6 Related work

With the popularization of social networks and availability of large amounts of relationship data in the form of graphs, graph data management and mining became an important area of research and development. A survey can be found here [1].

Graph representation is used frequently in many domains, such as social media and telcos. For example, we can model the relationships in a social network using graphs and finding communities in the graph [9] can facilitate targeted advertising. In the telco domain, CDRs can be used to capture the call relationships between people [25], and locating closely connected groups of people can be used for generating promotions. To handle the graph processing and management needs of an increasing number of applications in diverse domains, several graph processing and management systems have been introduced to handle large-scale graphs [11,12,15,18,21,23,24,29,31,38]. The primary goal of these systems is to manage large graphs and execute graph algorithms on them in an efficient and scalable manner.

In this work, we focus on disk-based graph management systems [15,26]. Unlike relational data, graphs are semi-structured in nature. Thus, storing and accessing graph data using secondary storage requires new solutions that can provide locality of access for graph processing workloads. In the literature there are several works which try to increase efficiency of graph management systems, like [14,28].

One of the primary contributions of our work is the scalable block formation algorithm used to generate locality-aware blocks by storing close vertices in the same blocks as much as possible. A relevant work in this area is the disk layout techniques proposed by Hoque and Gupta called Bondhu [14]. Bondhu [14] presents a strategy for storing a social graph on disk. It uses the community structures within the social graph as a placement strategy. Using this strategy, the disk layout is optimized, so that graph traversals can be performed using fewer I/O operations. Unlike Bondhu, ICBL is a distributed graph layout algorithm (based on MR) that can scale to large graphs.

In [28], Nodine et al. study the graph search problem for large graphs that cannot fit into the main memory by trying to use blocks on disk efficiently. As part of their work, it is shown that optimizing the block layout and access increases the performance of searching complete d-ary trees and d-dimensional grid graphs.

In [10], Gedik and Bordawekar have proposed a system for temporal storage and querying of evolving interaction graphs. In this work they proposed several online block formation algorithms that are used to reduce the I/O required to answer queries. Besides, they have proposed and applied several locality metrics to analyze graph blocks. In contrast to our work, their graphs are not relationship graphs, but instead append-only interaction graphs with a temporal aspect. As a result, their algorithms are streaming in nature.

GBASE [15] is a disk-based graph management system. It is related to our work in the sense that it is a Map/Reduce [7] based large-scale graph management system. It employs a graph storage method that relies on block compression to efficiently store homogeneous regions of graphs, and a grid-based technique to efficiently place blocks into files. However, the system is not optimized for locality-awareness.

In [2], Akyurek and Salem describe an adaptive technique for reducing disk seek times. To achieve this goal they copy a number of frequently referenced disk blocks to a reserved area near the middle of the disk from their current locations. Block rearrangement is related with our work, because we also need to arrange and order graph blocks on disk to achieve good performance. In [2], the arrangement of blocks are done based on block access frequencies and in our work we do it based on block similarities.

BORG [3] is a self-optimizing layer in the storage stack. It reorganizes data on disk by looking at access patterns. BORG aims to optimize read and write traffic dynamically by making them more sequential. This work is relevant with ours, in which we aim to organize locality-aware blocks of a graph on disk and make reads more sequential.

TurboGraph [13] is designed as a single PC graph processing system. It leverages the advantages of low latency and random I/O capabilities of SSDs. Although TurboGraph performs really well on SSD based disks, due to its parallel random I/O dependent design, it performs poorly on conventional magnetic disks.

In [37], Xie et al. propose a novel block-oriented computation model. In their model, computations are performed by iterating over locality-aware blocks. Although their computation model is based on the vertex-centric programming abstraction, instead of executing one vertex at a time they execute one block at a time and achieve good cache performance.

Neo4j [26] is a commercial disk-based graph management system. Although Neo4j implements optimizations such as indexing and caching, its on-disk graph layout can be improved to increase query performance. In this work, we have shown that locality-aware layouts generated by ICBL can be used to improve Neo4j's query performance by a factor of 2 or more.

In [8], Dominguez-Sal et al. study the characteristics of the graphs which are essential for benchmarks, and also the characteristics of the queries that are important in graph analysis applications. Their study has helped us determine graph characteristics that are useful for parameter selection in our experimental study.

7 Conclusion

We have developed a scalable method called ICBL that generates locality-aware disk blocks for large graphs. ICBL uses a series of MR jobs to divide the graph into disk blocks that contain sub-graphs with high locality. Furthermore, ICBL orders the resulting blocks on the disk to further reduce non-local accesses. ICBL makes the disk layout generation scalable, so that large-scale graphs can be divided into disk blocks using distributed processing. We proposed evaluation metrics for measuring the efficacy of the ICBL disk layout technique and presented an experimental evaluation showcasing its running time scalability, layout quality, as well as the effectiveness of automatic parameter tuning for ICBL. We demonstrated that ICBL is an effective disk layout technique for large-scale graphs and it increases the performance of disk-based graph management systems, such as Neo4j, by increasing the locality of access of disk blocks for common graph queries.

References

1. Aggarwal, C., Wang, H.: Graph data management and mining. In: Aggarwal, C. (ed.) A Survey of Algorithms and Applications. Springer, Berlin (2010)
2. Akyurek, S., Salem, K.: Adaptive block rearrangement. ACM Trans. Comput. Syst. **13**(2), 89–121 (1995). doi:10.1145/201045.201046
3. Bhadkamkar, M., Guerra, J., Useche, L., Burnett, S., Liptak, J., Rangaswami, R., Hristidis, V.: BORG: block-reorganization for self-optimizing storage systems. In: Proceedings of the 7th Conference on File and Storage Technologies, pp. 183–196 (2009)
4. Boldi, P., Vigna, S.: The WebGraph framework I: compression techniques. In: Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004), pp. 595–601 (2004)
5. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: Proceedings of the 20th International Conference on World Wide Web (2011)
6. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: Fourth SIAM International Conference on Data Mining (2004)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Symposium on Operating System Design and Implementation (OSDI), pp. 137–150 (2004)
8. Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, V., Baleta, P., Larriba-Pey, J.: A discussion on the design of graph database benchmarks. In: Nambiar, R., Poess, M. (eds.) Performance Evaluation, Measurement and Characterization of Complex Systems. Springer, Berlin (2011)
9. Fortunato, S.: Community detection in graphs. Phys. Rep. **483**(3–5), 75–174 (2009)
10. Gedik, B., Bordawekar, R.: Disk-based management of interaction graphs. IEEE Trans. Knowl. Data Eng. **26**(11), 2689–2702 (2014)
11. Giraph: Apache Giraph. http://www.giraph.apache.org/. Accessed June 2015
12. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Symposium on Operating System Design and Implementation (OSDI), pp. 17–30 (2012)
13. Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 77–85 (2013)
14. Hoque, I., Gupta, I.: Disk layout techniques for online social network data. IEEE Comput. **16**(3), 24–36 (2012)
15. Kang, U., Tong, H., Sun, J., Lin, C.Y., Faloutsos, C.: GBASE: a scalable and general graph management system. In: ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp. 1091–1099 (2011)
16. Karypis, G., Kumar, V.: Multilevel graph partitioning schemes. In: International Conference on Parallel Processing (ICPP), pp. 113–122 (1995)
17. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: WWW'10: Proceedings of the 19th International Conference on World Wide Web, pp. 591–600 (2010)
18. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Symposium on Operating System Design and Implementation (OSDI), pp. 31–46 (2012)
19. Lasalle, D., Karypis, G.: Multi-threaded graph partitioning. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 225–236 (2013)
20. Leskovec, J., Krevl, A.: SNAP datasets: Stanford large network dataset collection (2015). http://www.snap.stanford.edu/data
21. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012). doi:10.14778/2212351.2212354

22. MacQueen, J.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics, pp. 281–297 (1967)
23. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: ACM International Conference on Management of Data (SIGMOD), pp. 135–146 (2010)
24. Mondal, J., Deshpande, A.: Managing large dynamic graphs efficiently. In: ACM International Conference on Management of Data (SIGMOD), pp. 145–156 (2012)
25. Nanavati, A.A., Siva, G., Das, G., Chakraborty, D., Dasgupta, K., Mukherjea, S., Joshi, A.: On the structural properties of massive telecom call graphs: findings and implications. In: ACM International Conference on Information and Knowledge Management (CIKM), pp. 435–444 (2006)
26. Neo4j: Neo4j open source graph database (2015). http://www.neo4j.org/
27. Newman, M.: Power laws, Pareto distributions and Zipf's law. Contemp. Phys. **46**(5), 323–351 (2005). doi:10.1080/00107510500052444
28. Nodine, M.H., Goodrich, M.T., Vitter, J.S.: Blocking for external graph searching. Algorithmica **16**(2), 181–214 (1996)
29. Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., Haridasan, M.: Managing large graphs on multi-cores with graph awareness. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, pp. 4–4 (2012)
30. Rajaraman, A., Ullman, J.D.: Data mining. In: Mining of Massive Datasets, pp. 1–17. Cambridge University Press, Cambridge (2011)
31. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: ACM International Conference on Management of Data (SIGMOD) (2013)
32. Siek, J.G., Lee, L.Q., Lumsdaine, A.: Boost Graph Library. The User Guide and Reference Manual. Addison-Wesley, Boston (2002)
33. Simmhan, Y., Kumbhare, A., Wickramaarachchi, C., et al.: Goffish: a sub-graph centric framework for large-scale graph analytics. In: European Conference on Parallel Processing (Euro-Par), pp. 451–462 (2015)
34. Steinhaus, R.: G-Store: a storage manager for graph data. Master's Thesis, University of Oxford (2011)
35. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From think like a vertex to think like a graph. Proc. Very Large Databases Conf. **7**(3), 193–204 (2013)
36. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 409–410 (1998)
37. Xie, W., Wang, G., Bindel, D., Demers, A., Gehrke, J.: Fast iterative graph computation with block updates. Proc. Very Large Databases Conf. **6**(14), 2014–2025 (2013). doi:10.14778/2556549.2556581
38. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: a resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems, pp. 2:1–2:6 (2013)
39. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: a block-centric framework for distributed computation on real-world graphs. Proc. Very Large Databases Conf. **7**(14), 1981–1992 (2014)