Towards Effective Partition Management for Large Graphs

Shengqi Yang Xifeng Yan Bo Zong Arijit Khan
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110, USA
{sqyang, xyan, bzong, arijitkhan}@cs.ucsb.edu

ABSTRACT

Searching and mining large graphs today is critical to a variety of application domains, ranging from community detection in social networks, to de novo genome sequence assembly. Scalable processing of large graphs requires careful partitioning and distribution of graphs across clusters. In this paper, we investigate the problem of managing large-scale graphs in clusters and study access characteristics of local graph queries such as breadth-first search, random walk, and SPARQL queries, which are popular in real applications. These queries exhibit strong access locality, and therefore require specific data partitioning strategies. In this work, we propose a Self Evolving Distributed Graph Management Environment (Sedge), to minimize inter-machine communication during graph query processing in multiple machines. In order to improve query response time and throughput, Sedge introduces a two-level partition management architecture with complimentary primary partitions and dynamic secondary partitions. These two kinds of partitions are able to adapt in real time to changes in query workload. Sedge also includes a set of workload analyzing algorithms whose time complexity is linear or sublinear to graph size. Empirical results show that it significantly improves distributed graph processing on today's commodity clusters.

Source Code: `http://www.cs.ucsb.edu/~xyan/Sedge`

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Graph and tree search strategies; H.2.4 [**Database Management**]: Systems

General Terms

Algorithms, Performance

Keywords

Graph, Partitioning, Graph Query Processing, RDF, Distributed Computing

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.

1. INTRODUCTION

Large scale, highly interconnected networks pervade both our society and the information world around us [23, 29]. Online social networks capture complex relationships among millions of users. HTTP links connect billions of documents on the Web. Synthesized graphs are available from genome sequence alignment and program traces. The graphs of interest are often massive with millions, even billions of vertices, making common graph operations computationally intensive. In the presence of data objects associated with vertices, it is clear that graph data can easily scale up to terabytes in size. Moreover, with the advance of the *Semantic Web*, efficient management of massive *RDF* data is becoming increasingly important as Semantic Web technology is applied to real-world applications [1, 3]. The recent *Linked Open Data* project has published more then 20 billion RDF triples [14]. Although the RDF data is generally represented in triples, the data inherently presents graph structure and is therefore interlinked. Not surprisingly, the scale and the flexibility rise to the major challenges to the RDF graph management.

The massive scale of graph data easily overwhelms memory and computation resources on commodity servers. Yet online services must answer user queries on these graphs in near real time. In these cases, achieving fast query response time and high throughput requires partitioning/distributing and parallel processing of graph data across large clusters of servers. An appealing solution is to divide a graph into smaller partitions that have minimum connections between them, as adopted by Pregel [27] and SPAR [32]. As long as the graph is clustered to similar-size partitions, the workload of machines holding these partitions will be quite balanced. However, the assumption becomes invalid for *local graph queries* when they are concentrated on a subset of vertices (hotspots), e.g., find/aggregate the attributes of h-hop neighbors around a vertex, calculate personalized PageRank [18], perform a random walk starting at a vertex, and calculate hitting time. When these queries are not uniformly distributed or hitting partition boundaries, we will either have an imbalance of workload or intensive inter-machine communications. A good graph partition management policy should consider these situations and adapt dynamically to changing workload.

There could be three kinds of query workload in graphs. For random access or complete traversal of an entire graph shown in Figure 1(a), a static balanced partition scheme might be the best solution. For queries whose access is bounded by partition boundaries, as shown in Figure 1(b),

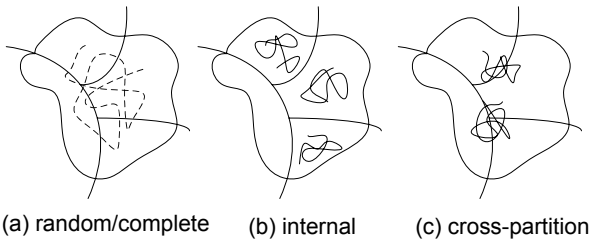


Figure 1: Query Access Pattern

they shall be served efficiently by the balanced partition scheme too. However, if there are many graph queries crossing the partition boundaries shown in Figure 1(c), the static partition scheme might fail due to inter-machine communications. *One partition scheme cannot fit all.* Instead, one shall generate multiple partitions with complementary boundaries or new partitions on-the-fly so that these queries can be answered efficiently.

Graph partitioning is a hard and old problem, which has been extensively studied in various communities since 1970s [22, 16, 31, 20]. Graph partitioning is also widely used in parallel computing (e.g., [15]). The best approaches often depend on the properties of the graphs and the structure of the access patterns. Much of the previous work has focused on graphs arising from scientific applications (meshes [12], etc) that have a different structure than social networks and RDFs focused in this study, where well-defined partitions often do not exist [25]. In this study, our focus is not to design new graph partition algorithms, but to adjust partitions to serve queries efficiently. We design a **S**elf **E**volving Distributed **Gra**ph **M**anagement **E**nvironment (Sedge). While Sedge adopts the same computation model and programming APIs of Pregel [27], it emphasizes graph partition management, which is the key to query performance. It adds important functions to support overlapping partitions, with the goal of minimizing inter-machine communication and increasing parallelism by dynamically adapting graph partitions to query workload change.

Our Contributions. A major contribution of this study is an examination of an increasingly important data management problem in large-scale graphs and the proposal of a graph partition management strategy that supports overlapping partitions and replicates for fast graph query processing. Dynamic graph partitioning and overlap graph partitioning were widely investigated before (e.g., [36]). However, few methods study how to adapt partitions to satisfy dynamic query workload in social and information networks. We addressed this issue and proposed Sedge, a workload driven method to manage partitions in large graphs. We eliminate a constraint in Pregel [27] that does not allow duplicate vertices in partitions. This constraint makes it difficult to handle skewed query workload. It is able to replicate some regions of a graph and distribute them in multiple machines to serve queries in parallel. For this goal, we develop three techniques in Sedge: (1) Complementary Partitioning; (2) Partition Replication; and (3) Dynamic Partitioning. Complementary Partitioning is to find multiple partition schemes such that their partition boundaries are different from one another. Partition replication is to replicate the same partitions in multiple machines to share the workload on these partitions. Dynamic Partitioning is to construct new partitions to serve cross-partition queries locally. In order to perform dynamic partitioning efficiently, we propose an innovative technique to profile graph queries. As manifested later, it is too expensive to log all of the vertices accessed by each query. We introduced the concept of color-blocks and coverage envelope to bound the portion of a graph that has been accessed by a query. An efficient algorithm to merging these envelopes to formulate new partitions is thus developed. The partition replication and dynamic partitioning are together termed on-demand partitioning since the two techniques are primarily employed during the runtime of the system to adapt evolving queries. Additionally, a two-level partition architecture is developed to connect newly generated partitions with primary partitions.

We implement Sedge based on Pregel. However, the concepts proposed and verified in this work are also valid to other systems. The performance of Sedge is validated with several large graph datasets as well as a public *SPARQL* performance benchmark. The experimental results show that the proposed partitioning approaches significantly outperform the existing approach and demonstrate superior scaling properties.

2. RELATED WORK

Graph partitioning is an important problem with extensive applications in many areas, including circuit placement, parallel computing and scientific simulation. Large-scale graph partitioning tools are available, e.g. METIS [20], Chaco [16], and SCOTCH [31], just to name a few. This study is not to propose a new graph partitioning algorithm. Instead, it is focused on a workload driven method to manage partitions in large graphs.

Distributed memory systems in super-computing is able to process large-scale linked data, e.g., [21, 28]. These systems could map shared data into the address space of multiple processors. They are usually very general, supporting random memory access that has less locality than the graph queries introduced in this work, thus could not benefit from query locality. Malewicz et al. [27] introduced Pregel, which could run graph algorithms in a distributed and fault-tolerant manner. Logothetis et al. [26] introduced a generalized architecture for continuous bulk processing (CBP) that is good for building incremental applications in large datasets including graphs. Najork proposed the scalable hyperlink store, SHS [28]. SHS studied several key issues in large graph processing: real-time response, graph compression, fault tolerance, etc. Our study touches another aspect on managing partitions to fit workload changes. Kang et al. [19] developed a peta-scale graph mining system, *PEGASUS*, built on the top of the Hadoop platform. PEGASUS proposed and optimized iterative matrix-vector multiplication operators. The difference between Pregel and MapReduce can be referred to [27]. In this work, we implement and leverage the computing environment provided by Pregel, but focus on graph partition management, not optimization techniques for specific algorithms. COSI [5] is a framework that is able to partition very large social networks according to query history. Such work is optimized for static query workload and hence cannot be readily applied to dynamic query workload. Pujol et al. [32] developed a social partitioning and replication middle-ware, *SPAR*, to achieve data locality while minimizing replication. SPAR aims to opti-

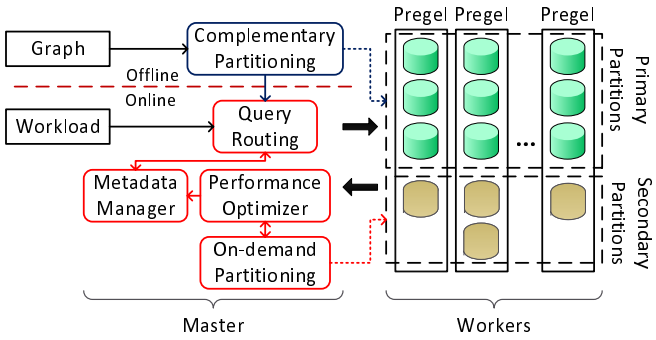


Figure 2: Sedge: System Architecture

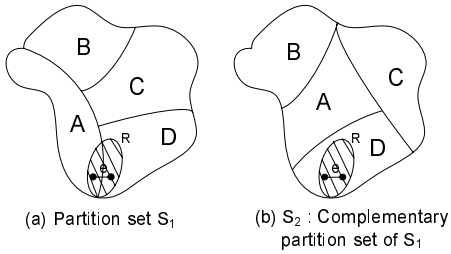


Figure 3: Complementary Partitioning: e is a cross partition edge in S_1 but not in S_2.

mize performance based on social network structures, e.g., communities, while our system develops partitioning techniques that adapt to query workload change. As discussed before, network structures might not reflect actual query workload. In addition to in-memory solutions, Nodine et al. [30] considered the problem of using disk blocks efficiently in searching graphs that are too large to fit in memory. The idea of using redundant blocks is related to complementary partitioning proposed in Sedge.

Distributed query processing has also been studied on semistructured data [35, 7], relational data [10] and RDF [17]. The key technique is minimizing data movement by partial evaluation, hybrid shipping, two-phase optimization and replication (see [24] for a survey). Additionally, as the emerging of Semantic Web, more and more data sources on the Web are organized in the RDF model and linked together. With the observation of the heterogeneity and scalability challenges existing in the management of RDF data, innovative data schemas have been proposed. One of the widely used techniques has been termed the *property table* [6, 37]. The technique is to cluster subjects sharing similar properties/predicates. Another technique, *vertical table* [1], is to vertically partition the schemas on property value. Efficient RDF data management is still an open problem and has not been addressed thoroughly.

3. SYSTEM DESIGN

Many applications [32, 10] employ graph partitioning methods for distributed processing. Unfortunately, real life networks such as social networks might not have well-defined clusters [25], indicating that many cross-partition edges could exist for any kind of balanced partitions. For queries that visit these edges, the inter-machine communication latency will affect query response time significantly. To alleviate this problem, we propose Sedge, which is based on multi partition sets (Figure 2).

Sedge is designed to eliminate the inter-machine communication as much as possible. As shown in Figure 2, the offline part first partitions the input graph in a distributed manner and distributes them to multiple workers. It creates multiple partition sets so that each set runs independently. Pregel [27] is a scalable distributed graph processing framework that works in a bulk synchronous mode. Pregel is used as a computing platform that is able to execute local graph queries. There are various kinds of local graph queries including breadth-first search, random walk, and SPARQL queries. Unlike many graph algorithms, a local query usually starts at one vertex and only involves a limited number of vertices (termed *active vertice*). In each iteration, a Pregel instance only accesses active vertices, thus eliminating many synchronous steps. Section 6 will discuss synchronization for the queries with writes and updates.

The online part collects statistical information from workers and actively generates and removes partitions to accommodate the changing workload. Therefore the set of online techniques built in Sedge must be very efficient to minimize overhead. Our study is focused on partition management. For fault-tolerance and live partition migration with ACID properties, detailed explorations of these issues are given in [27, 11] and similar techniques can be applied here. In the following discussion, we overview major components including complementary partitioning, on-demand partitioning, the mechanism to connect primary and secondary partitions, the meta-data to facilitate query routing and performance optimizer.

3.1 Graph Partitioning

DEFINITION 1 (GRAPH PARTITIONING). *Given a graph $G = (V, E)$, graph partitioning, C, is to divide V into partitions $\{P_1, P_2, \ldots, P_n\}$ such that $\cup_i P_i = V$, and $P_i \cap P_j = \emptyset$ for any $i \neq j$. The edge cut set E_c is the set of edges whose vertices belong to different partitions.*

Graph partitioning needs to achieve dual goals. On the one hand, in order to achieve the minimum response time, the best partitioning strategy is to split the graph using the minimum cut. On the other hand, taking the system throughput into consideration, the partitions should be as balanced as possible. This is exactly what the normalized cut algorithm can do [20]. Techniques derived from graph compression, e.g., [4] can also be applied here. However, partitioning a graph using a random hash function might not work very well.

Complementary Partitioning is to repartition a graph such that the original cross-partition edges become internal ones. Figure 3(b) shows an example of complementary graph partitions of Figure 3(a). In the new partition set, the queries (shaded area R) on original cross-partition edge, e, will be served within the same partition. Therefore, the new partition set can handle graph queries that have trouble in the original partition set. If there is room to hold both S_1 and S_2 in clusters, for a query Q visiting the shaded area R in S_1, the system shall route it to S_2 to eliminate communication cost. Meanwhile, the new partition set can also share the workload with original partition set. This complementary partitioning idea can be applied multiple times to generate a series of partition sets. We call each partition set

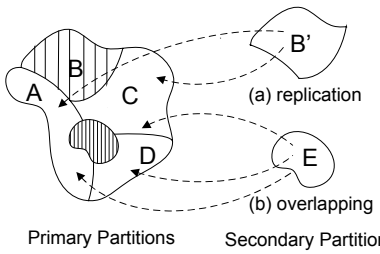


Figure 4: Two-Level Partition Architecture: Secondary partition B' on the top-right is a replicate of primary partition B. Secondary partition E covers the shaded region that crosses primary partition A, C and D.

a "*primary partition set*." Each primary partition set is self complete, where a Pregel instance can run independently.

Primary partition set can serve queries that are uniformly distributed in the graph. However, they are not good at dealing with unbalanced query workload: queries that are concentrated in one part of the graph. It will be necessary to either create a replicated partition (Figure 4(a)) or generate a new overlapping partition (Figure 4(b)) in an idle machine so that the workload can be shared appropriately. This strategy, called **On-demand Partitioning**, will generate new partitions online. These add-on partitions, called "*secondary partitions*", could last until their corresponding workload diminishes.

3.2 Two-Level Partition Management

Given many primary/secondary partitions, it is natural to inquire how to manage these partitions. Here we propose the concept of **Two-Level Partition Management**. Figure 4 depicts one example, where there are intensive workloads on two shaded areas. Based on a primary partition set, $\{A, B, C, D\}$, two secondary partitions, B' and E, are created to share the unbalanced workload on primary partitions. Since the vertices in secondary partitions are the duplicates of vertices in primary partitions, some of the vertices might connect to the vertices in primary partitions. Therefore it is necessary to maintain the linkage between vertices in secondary partitions and those in primary partitions. In our design, the linkage is only recorded in secondary partitions. It is not necessary to maintain such links in primary partitions. For example, for partition B', it has to maintain the linkage to A and C. While for A and C, they only maintain links to B, but not to B'.

During the runtime, each primary partition set and the corresponding secondary partitions are maintained by a Pregel instance that is running on a set of worker machines as indicated in Figure 2. Multiple isolated independent Pregel instances are coordinated by meta-data management.

3.3 Meta-data Management

Meta-data is maintained by both the master and the Pregel instances. As in Figure 2, the **meta-data manager** in the master node maintains the information about each live Pregel instance and a fine-grained table mapping vertices to the Pregel instances. An index mapping vertices to partitions is also maintained by each live Pregel instance. This two-level indexing strategy is used to facilitate fast **query routing**. Specifically, when a query is issued to the system, the routing component first checks the vertex table maintained by the master. The index entry maps the vertex id to the Pregel instance which can most efficiently execute the query. After the query is routed to a particular Pregel instance, it is the duty of the vertex index maintained by the Pregel instance to decide to which partition the query should be forwarded. The detailed techniques of indexing vertices and routing queries will be discussed in Section 6.

In order to facilitate different kinds of queries, in addition to vertex index, it is desirable to design indices for the attributes of vertices and edges. Efficient decentralized/distributed indexing techniques, such as [34], have come to the fore in recent years. However, this topic is beyond the scope of this work.

3.4 Performance Optimizer

The **Performance Optimizer** continuously collects runtime information from all the Pregel instances via daemon processes and characterizes the execution of the query workload, such as vertex access times of each partition, and the number of cross-machine messages/queries. The optimizer can update the meta-data maintained by the master and evoke on-demand partitioning routine as the workload varies. It is notable that although we depict the on-demand partitioning as a component on the master side in Figure 2, the routine is actually executed by the Pregel instance on the worker side in a distributed manner. Therefore the overhead of on-demand partitioning will be isolated and not affect the performance of other Pregel instances.

4. COMPLEMENTARY PARTITIONING

Complementary partitioning is to find multiple partition sets such that their partition boundaries do not overlap. Formally, we define the problem as:

Given a partition set $\{P_1, P_2, ..., P_k\}$ on G and the cut edges $E_c = \{e_1, e_2, ..., e_i\}$. The problem is to partition G into a new partition set $\{P'_1, P'_2, ..., P'_k\}$ satisfying the same partitioning criteria (e.g., minimum cut) such that the new cut edges do not overlap with E_c.

If we want to exclude more edges, E_c could be expanded to include edges near the original cut edges. Without loss of generality, we assume G is an undirected graph with unit edge weight. X is an $n \times k$ matrix, defined as follows,

$$x_{ij} = \begin{cases} 1 & v_i \in V(P_j), \\ 0 & otherwise. \end{cases}$$

X gives a k-partition set of G. Furthermore, we define the following constraints on X: (1) *full coverage and disjoint*: $X\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ is a all-ones vector with appropriate size; (2) *balance*: $X^T\mathbf{1} \leq \mathbf{m}$, where $m_i = (1+\sigma)\lceil \frac{n}{k} \rceil$. m_i is a rough bound of partition size; σ controls the size balance. (3) *edge constraint*: $tr\ X^T\mathcal{W}X = \mathbf{0}$, where $\mathcal{W} = (w_{ij})$ is defined as an edge restrictive $n \times n$ Laplacian matrix. Given the edge set E_c, if $e_{ij} \in E_c$, $w_{ij} = -1$, otherwise $w_{ij} = 0$. Additionally, $w_{ii} = -\sum_{j \neq i} w_{ij}$. The complementary partitioning problem can be described below:

$$\begin{aligned} minimize \quad & \frac{1}{2} tr\ X^T \mathcal{L} X \\ s.t. \quad & X \text{ is binary} \\ & X\mathbf{1} = \mathbf{1}, X^T\mathbf{1} \leq \mathbf{m} \\ & tr\ X^T\mathcal{W}X = \mathbf{0} \end{aligned} \tag{1}$$

where $\mathcal{L} = (l_{ij})$ is a $n \times n$ Laplacian matrix. By definition, if $e_{ij} \in E(G)$, $l_{ij} = -1$, otherwise $l_{ij} = 0$ and $l_{ii} = -\sum_{j \neq i} l_{ij}$. The objective function gives the overall cost of the cut edges with respect to a particular assignment of X.

The above problem is a nonconvex quadratically constrained quadratic integer program ($QCQIP$). We rewrite the problem formulation so that we can reuse the existing balanced partitioning algorithms:

$$\begin{aligned} \textit{minimize} \quad & tr \;\; X^T(\mathcal{L} + \lambda\mathcal{W})X \\ \textit{s.t.} \quad & X \textit{ is binary} \\ & X\mathbf{1} = \mathbf{1}, X^T\mathbf{1} \leq \mathbf{m} \end{aligned} \tag{2}$$

This new definition drops edge constraint in (1) and incorporate it into the objective function using a weighting factor λ on the cut edges. By changing the value of λ, we are able to control the overlap of the existing edge cut and the new edge cut generated by the complementary partition set. It also provides a scalable solution: Given the cut edges of the existing partition sets, we increase their weight by λ and then run balanced partitioning algorithms such as METIS [20] to perform graph partitioning.

The value of λ plays a critical role. Let the edge cut of the complementary partition set be E_c'. If its value is small, the partitioning algorithm can not distinct the cut edges with the others. On the other hand, if the value is too large, the algorithm might have to cut significantly more edges in order to completely avoid the existing edge cut. That is, E_c' might be much larger than E_c, which is not good too. In our implementation, we set $\lambda = 2^k$ and experiment different k with a set of simulated graph queries. For each k, we check the ratio $\beta = \frac{|E_c'| - |E_c|}{|E_c|}$. It was observed that when $k = 4$ and $\beta \leq 0.1$, the obtained partition set can achieve good performance.

Another possible technique for complementary partitioning is to delete all the edges in E_c first and then run classic partitioning algorithm. We argue that this approach doesn't work since (1) edge deletion destroys the structure of the graph, and thus the new result may probably not reflect the real connections among the graph partitions; (2) in order to preserve a good partition schema, i.e., minimum cut, in complementary partitioning, some of the edges should be included in the edge cut repeatedly.

The heuristic algorithm can be applied multiple times to generate a series of complementary partition sets, each of which try to partition the graph such that the boundary edges in one partition set will be internal edges in another partition set. With multiple partition sets, for each vertex u, there could be several partitions $P_1, P_2, \ldots, P_l$ to handle queries submitted to u. Queries should be routed to a partition where u is far away from partition boundaries. We define such a partition as a *safe partition* for vertex u. As soon as a new complementary partition set is generated, we can obtain the safe partitions for the vertices, especially those on the boundary of the original partitions.

Remark. There are some extreme cases, e.g., complete graph, where no complementary partition schema exists. However, for large graphs with small dense substructures, we can continuously perform complementary partitioning. In reality, due to space limitation, we can only afford a few sets of complementary partitions, and resort to on-demand partitioning algorithms to handle skewed query workloads that target some hotspots.

5. ON-DEMAND PARTITIONING

In the processing of many graph queries, primary partitions could have hotspots that are frequently visited. The queries heading to these partitions will suffer longer response time. There are two kinds of query hotspots: (1) internal hotspots that are located in one partition; (2) cross-partition hotspots that are on the boundary of multiple partitions. We developed two partitioning techniques, *partition replication* and *dynamic partitioning*, to generate secondary partitions on demand to handle hotspots.

5.1 Partition Replication

Definition 2 (Partition Workload). *Given a graph G, a partition $P \subseteq G$, and a query set $Q = \{q_1, q_2, \ldots, q_m\}$, the query set of P, written $W(P)$, is the queries that have accessed at least one vertex in P. The internal query set of P, written $W_{int}(P)$, is the set of queries that only accessed vertices in P. The external (cross-partition) query set of P, written $W_{ext}(P)$, is equal to $W(P) - W_{int}(P)$.*

Given a partition P, when its internal workload ($W_{int}(P)$) becomes intensive, it will saturate the CPU cycles of the machine that holds P. One natural solution is to replicate P to P'. If there is an idle machine with free memory space, Sedge will send P' to that machine. Otherwise, it will find a slack partition and replace it with P'. A slack partition is a secondary partition with low query workload on it. By routing queries to P', the workload on P could be reduced.

5.2 Cross-partition Hotspots

When cross-partition hotspots exist, primary partitions have to communicate with each other frequently to answer cross-partition queries. Instead of replicating multiple partitions, it is better to generate new partitions that only cover cross-partition hotspots. The new partitions will not only share heavy workload, but also reduce communication overhead, thus improving query response time.

Hotspot Analysis. Before assembling a new partition, we need to find cross-partition hotspots first. Given a partition, we calculate a ratio $r = \frac{|W_{ext}(P)|}{|W_{int}(P)| + |W_{ext}(P)|}$ and resort to a hypothesis testing method to detect abnormal cross-partition query workload.

If a query is uniformly and randomly distributed over a partition P, we can calculate the probability of observing a cross-partition query in P by either doing a simulation or approximating it using the following external edge ratio, $p = \frac{|E_{ext}(P)|}{|E_{int}(P)| + |E_{ext}(P)|}$, where $|E_{ext}(P)|$ is the number of cross-partition edges between P and other partitions, and $|E_{int}(P)|$ is the number of internal edges. If r is significantly higher than p, it could be reasonably assumed that there are cross-partition hotspots in P. Let $n = |W_{int}(P)| + |W_{ext}(P)|$ and $k = |W_{ext}(P)|$. The chance to have $\geq k$ cross-partition queries is

$$Pr(x \geq k) = \sum_{i=k}^{n} \binom{n}{i} p^i (1-p)^{n-i}.$$

When $Pr(x \geq k)$ is very small (e.g., 0.01), it means there is an abnormal large number of cross-partition queries in P.

5.3 Track Cross-partition Queries

Besides detecting cross-partition hotspots, we need a method to track the trail of cross-partition queries and pack them

to form a new partition. It is intuitive to record each query in the form of its exact search path. However, it is not only space and time consuming for profiling, but also difficult to generalize. Instead we mark the search path of a cross-partition query with coarse-granularity units, *color-blocks*.

A **color-block** is a set of vertices $V_i \subset V$ where they are assigned with a unique color c_i. For any vertex $v \in V$, it has one and only one color. Using color-blocks, we are able to coarsen a graph with a much smaller number of units. To form color-blocks, we experimented on several algorithms, i.e., *nearest-k neighbors*, *neighbors within k-hops*, etc, and found that neighbors within 1-hop outperforms the others. Disjointed 1-hop color-blocks could be generated as follows: (1) randomly select one vertex, find its 1-hop neighbors, and form a color-block; (2) delete the vertices of this color-block; (3) repeat (1) and (2) until no vertex is left.

5.4 Dynamic Partitioning

[Query Profiling] Given a set $C = \{c_1, c_2, ..., c_n\}$ of color-blocks, we track the trail of a query with a subset of color-blocks, $L_j = \{c_{j_1}, c_{j_2}, ..., c_{j_l}\}$. Since these color-blocks will be grouped together later, it is not necessary to record the visiting order of color-blocks. L_j is termed an **envelope** of the query.

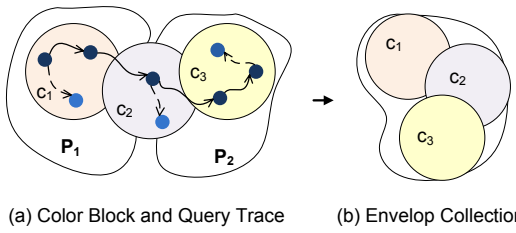


Figure 5: Color-block and Envelop Collection

By tracking cross-partition queries using color-blocks, each query can be profiled as an *envelope*. Figure 5 shows the relation among partitions, color-blocks and envelopes. Given a set of candidate envelopes, a partition cannot assemble all of them due to its space constraint. Herein we formulate the problem as an *envelopes collection* problem.

[Envelopes Collection] Given a partition with the storage capacity M, there are a set $L = \{L_1, ..., L_n\}$ of envelopes and a set $\bigcup_{j=1}^{n} L_j$ of m colors, each envelope L_j encapsulates a set $L_j = \{c_{i_1}, c_{i_2}, ..., c_{i_l}\}$ of colors and the size of color c_k is w_k. If $D \subseteq L$ and $R = \bigcup_{L_j \in D} L_j$, the objective is to find such a set D that maximizes $|D|$ with the constraint $\sum_{c_k \in R} w_k \leq M$, where M is the default partition size.

Envelopes collection is reminiscent of the *Set-Union Knapsack Problem*, which is a classic NP-complete problem. We propose a greedy algorithm based on the intuition that combining similar envelopes consumes less space than combining non-similar ones. Given two envelopes L_i and L_j, the overlap of their color-block sets is measured as the *Jaccard coefficient* $Sim(L_i, L_j) = \frac{|L_i \cap L_j|}{|L_i \cup L_j|}$. Given n envelopes, performing pair-wise similarity comparison is a procedure running in $O(n^2)$. To cope with this challenge, we employ a hash-based algorithm, called *Locality Sensitive Hashing* (LSH)[13] to perform similarity search in a provably sublinear time.

LSH is a probabilistic method that hashes items so that similar items can be mapped to the same buckets with high probability [13]. In our case, we adopt a LSH scheme called *Min-Hash* [9]. The basic idea of Min-Hash is to randomly permute the involved set of color-blocks and for each envelope L_i we compute its hash value $h(L_i)$ as the index of the *first* color-block under the permutation that belongs to L_i. It has been shown in [9] that if we randomly choose a permutation that is uniformly distributed, the probability that two envelopes will be mapped to the same cluster is exactly equal to their similarity. We use Min-Hash as a probabilistic clustering method that assigns a pair of envelopes to the same bucket with a probability proportional to the similarity between them. Each bucket is considered as a cluster and the envelopes within the same bucket are combined together.

Algorithm 1 Similarity-Based Greedy Clustering Algorithm

Input: Envelope set $L = \{L_i\}$
Output: New partition P
1: Initialize hash functions
2: **for** each $L_i \in L$ **do**
3: hash_value $= h(L_i)$
4: add L_i to C_{hash_value}
5: **end for**
6: $C = \{C_{hash\_value}\}$ for each $C_{hash\_value} \neq \emptyset$
7: **for** each cluster C_i in C **do**
8: $\rho[i] = |W(C_i)|/|C_i|$
9: **end for**
10: Sort clusters on ρ in descending order
11: cluster set $P = \emptyset$
12: Add clusters to P as many as possible, s.t., $size(P) \leq M$

[Partition Generation] After obtaining a set of independent clusters, each cluster is assigned with a benefit score, $\rho = \frac{|W(C)|}{|C|}$, to measure the quality of the cluster. Here $|W(C)|$ is the number of cross-partition queries denoted by all the envelopes in the cluster C (more accurately, the times of the color-blocks in C are accessed) and $|C|$ is the size of the cluster. We create an empty partition and iteratively assemble the cluster with the highest ρ at each step as long as the total size is no greater than the default partition size M.

Scalability issues. The greedy algorithm is outlined in Algorithm 1. For n envelopes, the complexity of Min-Hash clustering is $O(n)$ (lines 1-5) and the sorting runs in $O(mlog(m))$ (line 9) where m is the number of the clusters generated (line 6). In the worst case, combining the clusters needs $O(nm)$ (line 12). In total, the complexity of this greedy algorithm is $O(nm)$. There is still a concern that if n and m are large, this algorithm would lead to poor scalability. To cope with this challenge, we limit the growth of n and m in the following way. On one hand, we use a *sampling* method to constrain the size of n. For example, when the dynamic partitioning procedure is triggered, among a set of cross-partition queries we randomly select a number of queries as a sample to generate the new partition. On the other hand, we could coarsen the size of color-blocks by increasing the number of vertices included in these blocks. This will result in a color set much smaller than the vertex set. In the experiment, we show that these two methods collectively guarantee that the dynamic partitioning method works in an efficient way.

Discussion: Duplicate Sensitive Graph Query. As a design principle, primary partitions are disjointed: each

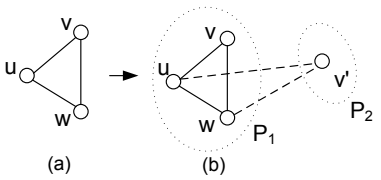


Figure 6: Duplicate Vertex

vertex only has one copy in the partitions. However, when secondary partitions exist, it is often the case that there are two copies v and v' for the same vertex. It might cause a potential issue, as illustrated in Figure 6. Figure 6(a) shows the original graph. In Figure 6(b), secondary partition P_2 is added and v' is a duplicate vertex v. Suppose we run the following algorithm to calculate the number of v's 2-hop friends :

[Method 1] *Starting at v', we send a message to its* 1*-hop friends and these friends send another message to their* 1*-hop friends. Each partition reports the number of vertices who received messages. Sum up the numbers.*

The above algorithm works correctly in primary partitions. However, for Figure 6(b), it will produce a wrong answer. Due to this complication, it is not straightforward to run queries correctly in secondary partitions. Fortunately, for many local graph queries, there are implementations that are not sensitive to overlapping partitions. If we change Method 1 slightly, it will work correctly.

[Method 2] *Starting at v', we send a message to its* 1*-hop friends and these friends send another message to their* 1*-hop friends. Each partition reports the vertices who received messages. Union the results by removing duplicates.*

Other graph queries such as random walk, personalized PageRank, hitting time and neighborhood intersection have implementations that are not sensitive to duplications. We call queries that can be correctly answered on overlapping partitions *Duplicate Insensitive Graph Queries*. If a duplicate sensitive graph query running on a secondary partition exceeds the boundary of the partition, the query will be terminated and restarted in a primary partition. In Sedge, the query routing component (described in the next section) maintains a vertex-partition fitness list for the start vertex of a query. It helps route the query to a partition that can serve it locally with high probability.

6. RUNTIME OPTIMIZATION

6.1 Query Routing

An incoming query arrives with at least one initial vertex. The master node dispatches the query to a Pregel instance according to the initiated vertex. As shown in Figure 3, if possible, a query shall be routed to a Pregel instance (PI for short) where its initiated vertex is in the safe region. Here, we devise a data structure in the master node to coordinate query routing:

- Instance Workload Table (IWT): $I \rightarrow W(I)$, where I is the ID of a PI and $W(I)$ is the workload of the PI.
- Vertex-Instance Fitness List (VFL): $v \rightarrow L_v\{I\}$, where $L_v\{I\}$ is an id list of the PIs.

Given a vertex v, the PIs where v is in safe region are ranked higher in VFL. Since some vertices, such as those with very high degree, might not be in any safe region, we assign a random order of PIs to their VFLs. During the runtime, the IWT is updated by the monitoring routine. Given a query, the algorithm routes the query to the first PI in its VFL that is not *busy* with respect to the IWT. Once the query is finished, if the query cannot be served locally in its assigned PI, the query fitness list will shift the PI to the end of the list. Since the number of Pregel instances is small, VFL is implemented using *bitset*. Bitset is an array optimized for space allocation: each element occupies only one bit. For example, it uses only 3 bits to represent up to 8 PIs. Our experimental results show that the simple greedy routing strategy can outperform random query routing significantly.

Vertex-Partition Mapping. In order to process queries, each Pregel instance needs to maintain the following tables to map vertices to partitions. All partitions are mapped onto unique IDs.

- Partition Workload Table (PWT): $P \rightarrow W(P)$, where P is the ID of a partition and $W(P)$ is the workload.
- Vertex-Primary Partition Table (VPT): $v \rightarrow P$, where P is a primary partition. Each vertex is mapped to one and only one primary partition.
- Partition-Replicates Table (PRT): $P \rightarrow \{S_R\}$, where $\{S_R\}$ are the identical replicates of P. For $\forall v \in P$, it may associate with several S_R.
- Vertex-Dynamic Partitions Table (VDT): $v \rightarrow \{S_D | v \in S_D\}$, where $\{S_D\}$ are the new partitions generated by the dynamic partitioning method.

Space complexity. Due to the limited number of partitions in practice, the size of the PWT and the PRT is negligible. VPT is $O(n)$, where n is the number of vertices in G. It only takes several gigabytes to store a VPT table for billions of vertices. The size of VDT depends on the number of vertices covered by the secondary partitions. Usually, the size is far smaller than $O(n)$.

In particular, each secondary partition is associated with one primary partition set from which it is created. When a secondary partition is generated or deleted, an entry in PRT or VDT needs to be updated accordingly. For K Pregel instances, we maintain their tables separately. That is, we will have K sets of PWT, VPT, PRT and VDT. These tables are stored in main memory.

6.2 Partition Workload Monitoring

The workload monitoring component in Sedge is built in the *optimizer* module (ref. to Figure 2). Report messages from all Pregel instances are sent to the master at the end of each period. Typically a report message from a Pregel instance I includes the number of the queries served in I (i.e., $W_{int}(I)$ and $W_{ext}(I)$), the total access times of the vertices ($\sum_{q \in W(I)} |V(q)|$), and the CPU run time of the machines holding I. These messages encode the workload information of Pregel instances. The master updates the IWT accordingly. Analogously, each Pregel instance collects runtime information of their partitions and calculates the ratio between the total access times of the vertices and the size of the partition and sorts the partitions based on the ratio. Then with respect to the threshold ratio, a partition can be marked as a *hot* or a *slack* one. The information is maintained in the PWT.

6.3 Partition Replacement

As discussed in Section 5, secondary partitions are generated to deal with query *hotspots*. In practice, the space that can be used to accommodate additional partitions is often limited. Therefore, it is unlikely to create as many secondary partitions as possible. At the same time, in real world applications, query hotspots may become "*slack*" ones after a period. This practical issue motivates a partition replacement scheme that replaces a slack secondary partition with a newly generated one. In Sedge, when a replacement is needed, we simply select the slackest secondary partition and replace it with the one newly generated.

6.4 Dynamic Update and Synchronization

Real world graphs usually change over time in terms of insertion and deletion of nodes and edges. Sedge can adapt to these dynamic changes. Here we take the update on one Pregel instance as an example. Since the information of a vertex can be obtained by referring to the vertex-partition map, edge insertion and deletion can be accomplished directly. For the insertion/deletion of edge (u, v), find the primary and secondary partitions of u and v, insert or delete the edge. To delete vertex v, one can retrieve all of its edges and delete them, and then retrieve all of partitions containing v and delete v. For insertion of vertex v and its edges, one can first locate a primary partition P where the majority of v's neighbors are located, and then add v to that partition. Meanwhile, update all of the replicates of P and then submit edge insertion requests. For vertex insertion and deletion, we also need to update the vertex-partition map, i.e., *VFL*, *VPT* and *VDT*. Note that the update should be applied to all the Pregel instances. When the insertion of vertices and the following edge insertions make a primary partition too big, we need to redo the partitioning from scratch. Additionally, when a query changes vertex values during its execution, the cost of keeping the vertex values in sync is usually quite high especially when there are many duplicates. In Sedge, we adopt a simple strategy: when a query changes a vertex value, a new update query is issued to all the corresponding partitions. An experiment in Section 7.2 demonstrates the efficiency of dynamic update in Sedge.

7. EXPERIMENTAL EVALUATION

The system is programmed in Java. We use a distributed version of METIS [20] to generate primary partitions. To evaluate Sedge on a diversified set of graphs and queries, we test datasets in two categories: RDF benchmarks and real graph datasets using different sets of graph queries. Our experiments are going to demonstrate that (1) Sedge is efficient and scalable, in comparison with the situation without partition management, and (2) the design of each component including complementary partitioning and on-demand partitioning is effective for performance improvement.

The experiments are conducted on a cluster with 31 computing nodes: each has 4 GB RAM, two quad-core 2.60GHz Xeon Processors and a 160 GB hard drive. Among these nodes, one serves as the master and the rest as workers. The cluster is connected by a gigabit ethernet. In each experiment, we perform three cold runs over the same experimental setting and report the average performance.

For each graph in the following experiments, we generate 5 complementary partition sets beforehand. We use CP_1 to denote the performance when only using the first primary partition set while CP_2, CP_3, CP_4 and CP_5 to denote the performance when using 2, 3, 4 and 5 partition sets, respectively. Each primary partition set consists of 12 primary partitions, which fill in 6 workers.

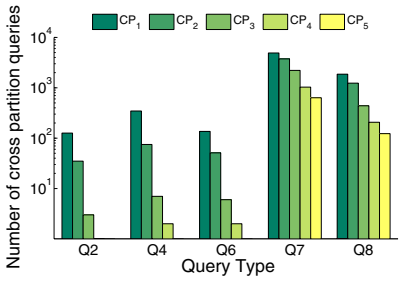


Figure 7: Number of cross-partition queries. The missing bars for the CP_4 and CP_5 of $Q2$, the CP_5 of $Q4$ and the CP_5 of $Q6$ correspond to the value of 0, i.e., the cross-partition query vanishes.

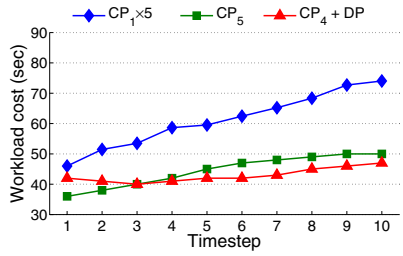
7.1 Evaluation with a SPARQL Benchmark

We first evaluate the system performance of Sedge on a SPARQL benchmark graph. SPARQL is an emerging standard for RDF. Efficient storage techniques for large scale RDF data and evaluation strategies for SPARQL are currently under exploration in the database community [33, 3]. In this experiment, we will illustrate that our partitioning techniques can improve SPARQL query execution significantly.

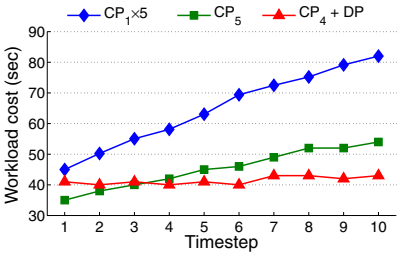
The SP^{2}Bench Benchmark [33] chooses the *DBLP* library as its simulation basis. It can generate arbitrary large RDF test data which mirrors vital real-world distributions found in the original DBLP data. Using the generator provided by [33], we create an RDF graph with $100M$ edges (11.24GB). It is a heterogenous graph with the subjects/objects as the vertices and the predicates as the links.

SP^{2}Bench provides 12 query templates, $Q1, Q2, \ldots, Q12$ that are delicately designed to capture all key features of the SPARQL query language. In this work, we select five categories in which the existing SPARQL engines have difficulties. These queries are listed in the *Appendix*. From the view of query operation, $Q6$ and $Q7$ encode the operations of OPTIONAL (akin to left outer joins) with FILTER and BOUND; from the view of access pattern, $Q2$ and $Q4$ contain two distinctive graph patterns, "*long path chains*" and "*bushy patterns*" [33]; $Q8$, extracting the *Erdös Number* of the authors, showcases the queries that concentrate on a "*hotspot*". We map the queries against specific vertices as the query starts and thereafter match the variables to the nodes or edges during the query execution.

In order to validate the complementary partitioning approach, we generate a workload with $10,000$ queries, which are the equal mixture of the 5 query types with randomly selected starts. The queries are routed automatically to the corresponding partitions with the assistance of the query routing module. We compare the performance by varying the number of the used primary partition sets. Figure 7 shows the effect of the approach. Note that the Y-axis is plotted in logarithmic scale to accommodate the significant differences in the number of queries that access at least two partitions. It is observed that by adding more complemen-



(a) Lifetime$_c$ = 2



(b) Lifetime$_c$ = 5

Figure 8: System performance with complementary partitioning and on-demand partitioning for evolving queries.

tary partition sets, the number of cross-partition queries can be dramatically reduced. It vanishes for Queries $Q2$, $Q4$ and $Q6$ when 4 or more complementary partition sets are used. A close look at the difference in the performance between the variants of query types reveals that $Q2$, $Q4$ and $Q6$ exhibit high locality. In contrast, $Q7$ and $Q8$ exhibit more complex access pattern. Figure 7 shows for the queries of $Q7$ and $Q8$, CP_5 outperforms CP_1 by up to almost one order of magnitude. The result suggests that our complementary partitioning is an effective way in response to cross-partition queries of various types. Figure 7 also shows, with respect to different queries, how the percentage of vertices in safe partitions changes when the number of complementary partition sets increases. For example, for $Q7$, the percentage of vertices in safe partitions increases from 50.9% (1 partition set) to 94.7% (5 complementary partition sets); and for $Q8$, it increases from 81.4% to 97.6%.

To demonstrate how Sedge responds to skewed workloads, we generate a synthetic evolving workload which contains 10 timesteps. In each timestep, the workload consists of $10,000$ queries which are the mixture of the 5 query types with equal number. To control the evolution of the workload, each query is assigned with a lifetime value. If the query is internal (finished within a partition), it has lifetime, $lifetime_I$; otherwise, it has lifetime, $lifetime_C$. When a query expires, it will restart in the next timestep with a new lifetime and a randomly selected start. Since random internal queries do not contribute to a skewed workload, we set $lifetime_I = 1$ for simplicity and vary the value of $lifetime_C$ in the following experiments. Note that when $lifetime_C > lifetime_I$, the number of cross-partition queries will increase gradually because more internal queries will become cross-partition queries than the reverse along the time.

We compare the approaches from two perspectives: complementary partitioning and on-demand partitioning. $CP_1 \times 5$ uses 5 static replicates of the first partition set (i.e., run five Pregel's independently, each with 1/5 workload), and CP_5 uses all the 5 complementary partition sets. Both of the two approaches use up 30 worker space. Note that we run these two settings only using Pregel instances where no query profiling (on-demand partitioning) is applied. $CP_4 + DP$ uses 4 complementary partition sets and employs the rest worker space for on-demand partitioning. To maintain a fair comparison, the number of secondary partitions can not exceed 12, the size of one partition set in our experiments.

Figure 8 reports the accumulated time cost of the query workload at each timestep with respect to the three approaches. The overhead of on-demand partitioning is also included in the workload cost. Figure 8(a) shows the performance of these approaches when $lifetime_C = 2$. The curve of CP_5 illustrates that the complementary partitioning technique significantly outperforms the static replication ($CP_1 \times 5$). The advantage becomes more obvious along with the accumulation of the cross-partition queries. It can also be seen that due to the generation of new secondary partitions, $CP_4 + DP$ outperforms CP_5 after timestep 3. When $lifetime_C = 5$, Figure 8(b) shows a similar result of the comparison between $CP_1 \times 5$ and CP_5 as in Figure 8(a). However, in Figure 8(a), $CP_4 + DP$ outperforms CP_5 noticeably after timestep 3 and the time cost almost remains steady. This is because when $lifetime_C = 2$, due to the dynamics of the queries, the system invokes on-demand partitioning more frequently (6 times) than that when $lifetime_C = 5$ (3 times).

7.2 Evaluation with Real Graph Datasets

Next, we evaluate the design of Sedge by testing the effectiveness of each component. We use another set of graphs and queries to show the broad usage of Sedge. Nevertheless, the same test can be conducted on SP^{2}Bench and similar results will be observed.

Web graph. It is a uk-2007-05 web graph data from http://webgraph.dsi.unimi.it [4], which is a collection of UK websites. It contains 30M vertices and 956M edges.

Twitter graph. The Twitter graph is crawled from *Twitter*, consisting of 40.1M users. There are 1.4B edges (including multi-edges) in this dataset. For simplicity, we aggregated the multi-edges and the associated attributes as one edge which represents several messages sent from one user to another at different time.

Bio graph. The Bio graph is a *de Bruijn Graph* built from a sample of mRNA. In this graph, vertices represent sub-sequences of DNA symbols with length of twenty one (a.k.a. *k-mer length*) and edges represent the adjacent relationships between vertices: the two vertices differ by a single symbol [38]. We collect 50M vertices and construct 68M edges. The resulting de Bruijn graph is like a tree.

Synthetic scale-free graph. The graph is generated based on R-MAT [8]. It consists of 0.2 billion vertices and 1.0 billion edges. The graph matches "*pow-law*" behaviors and naturally exhibits "*community*" structure.

Table 1 summarizes the size of the graphs, the time cost of building one primary (complementary) partition set, the size of the vertex-instance fitness list (*VFL*), and the size of the vertex-partition table (*VPT*). It can be seen that the auxiliary meta-data is much smaller than the graph it serves, only $0.5\% - 5\%$ of its size.

We use three classic local graph queries to experiment the performance: (1) *h-hop Neighbor Search (h-NS)*: the query

Table 1: Graph Datasets

Graph	Size (GB)	Partition (s)	*VFL* (MB)	*VPT* (MB)
Web	14.8	120	81.5	35.3
Twitter	24	180	109.0	45.4
Bio	13	40	135.9	55.3
Syn.	17	800	543.7	205

starts from a vertex v and does a breath-first search for all the vertices within h hops of v; (2) *h-step Random Walk (h-RW)*: the query starts at a vertex and at each following step jumps to one of its neighbors with equal probability. The query consists of h steps; (3) *h-step Random Walk with Restart (h-RWR)*: it is a *h-step random walk* query; but at each step it may return to its start vertex with p probability. We set $p = 10\%$ by default. For global graph algorithms like single-source shortest distance, Sedge could also support them. However, they are not the focus of this work.

We test the effectiveness of our proposed algorithms: *complementary partitioning*, *partition replication* and *dynamic partitioning*. Due to the space limitation, we first show the experiments on the *Web graph* with different test settings. For the other datasets, we get quite similar results. We will then give an evaluation of the system on the scalability, using all of the four graphs.

7.2.1 Complementary Partitioning

Figure 9 shows the effect of complementary partitioning in reducing the communication cost. In this experiment, we use CP_1 as the baseline (the result will not change if we replicate CP_1 five times) and test 10,000 *h-RWR* queries using different number of complementary partition sets. By varying the step of the *h-RWR*, it can be seen that the complementary partitioning method can reduce the inter-machine messages. As to queries with longer random walk, the performance of Sedge degrades. However, with more complementary partitions, e.g., CP_4 and CP_5, Sedge can still achieve good performance in message reduction.

7.2.2 Partition Replication

To evaluate the performance of partition replication on unbalanced workload, we randomly generate a workload with mixed queries, i.e., *3-NS*, *5-RW*, *5-RWR*, on a specific graph partition (denoted as P_1) and continuously increase the number of queries from 10,000 to 50,000. We run this changing workload under 3 different settings: (1) CP_1 (the baseline); (2) CP_1 and 1 replicate of P_1 (ref. as $CP_1 + P_S$); (3) CP_1 and 2 replicates of P_1 (ref. as $CP_1 + P_S \times 2$). Figure 10 shows the number of queries can be served per second (throughput) for each setting. It is observed that the throughput by using partition replication significantly outperforms that of no replication one. This is because the query workload on $P1$ is distributed and processed in parallel among the primary partition and its replicates.

7.2.3 Dynamic Partitioning

To test the performance of dynamic partitioning, we focus on queries that access multiple partitions. We randomly generate mixed cross-partition queries (*3-NS*, *5-RW* and *5-RWR*) and test the system performance by varying the number of queries from 10,000 to 50,000. We run Sedge with only one primary partition set (CP_1) as well as with one

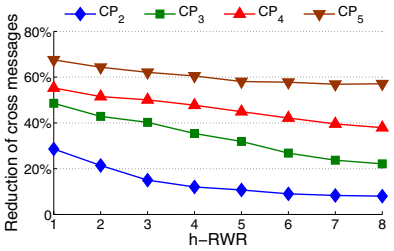


Figure 9: Complementary Partitioning: reduction of cross-partition messages. The x-axis shows the value of h, the number of walk steps.

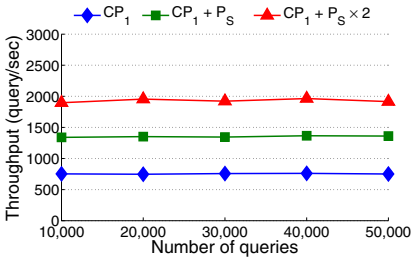


Figure 10: Partition replication: throughput

primary partition set and on-demand generated secondary partitions ($CP_1 + DP$), respectively.

Figure 11 shows the runtime cost of dynamic partitioning. It measures the run time of each stage to finish a dynamic partitioning process: *query profiling*, *envelopes collection* and *new partition generation*. The figure shows the cost per query by varying the number of cross-partition queries. For all the three stages, it is observed that the cost remains almost constant. Therefore the dynamic partitioning method is scalable with respect to the number of cross-partition queries.

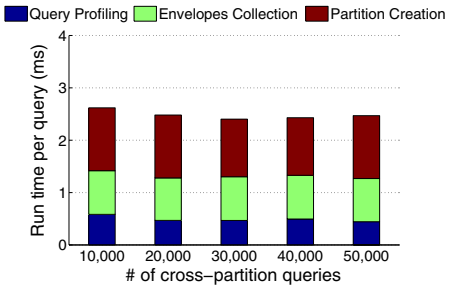


Figure 11: Dynamic Partitioning: runtime cost.

We next use the same query workload to test the effect of dynamic partitioning. Figure 12 shows the average response time by varying the number of cross-partition queries. Note that the response time here only indicates the query answering time. From the figure, we can observe the query response time is significantly improved compared to the static partitioning method. This also explains that our algorithms are effective for serving cross-partition queries. In the above experiments, Sedge uses slightly larger space with secondary partitions.

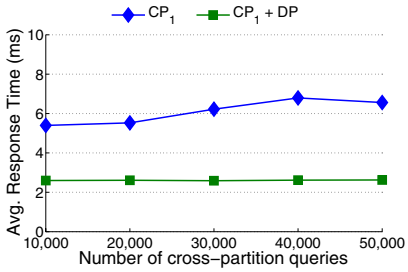


Figure 12: Dynamic partitioning: response time

7.2.4 *Scalability Evaluation*

Additionally, we test the capability of Sedge to handle intensive cross-partition queries. We generate five sets of query workload, each of which contains 100,000 random queries and set the percentage of the cross-partition queries as 0%, 25%, 50%, 75% and 100%, respectively. For this experiment, we use CP_1 as the baseline and demonstrate the performance of $CP_1 + DP$, where DP denotes secondary partitions generated by dynamic partitioning on demand. We employ 6 machines to hold CP_1 and assign additional machines gradually to accommodate the new partitions.

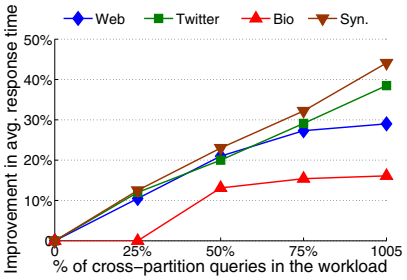


Figure 13: Cross-partition queries vs. Improvement ratio in avg. response time

Figure 13 shows the improvement ratio in average response time. In this figure, we plot the lift of the average response time by using on-demand partitioning compared with the baseline. The response time includes both the query answering time and the overhead of on-demand partitioning. As we increase the percentage of cross-partition queries, it can be seen that for all the four datasets, there is a significant improvement in average response time. In detail, however, we observe different improvement performance with respect to the changing workload. For the *Twitter graph* and *Synthetic graph*, the ratio increases constantly. This can be explained as follows. In these two graphs, there are many tightly connected substructures (*communities*). If these substructures are divided among multiple partitions, the cross-partition queries on them will visit these partitions frequently and as a result produce much inter-machine communication. In this case, by collecting the hot substructures together, our system can dramatically improve the efficiency. As for the *Bio graph*, it is a tree-like structure. Hence, the cross-partition query does not visit many partitions and the improvement in query response time is not remarkable when compared with the baseline method. The characteristics of the *Web graph* are between these two types.

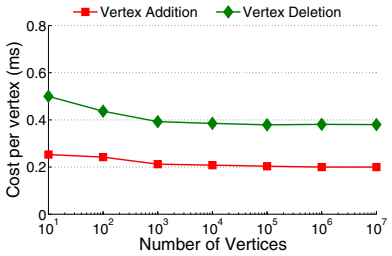


Figure 14: Dynamic Update/Synchronization Cost

7.2.5 *Dynamic Updates and Synchronization*

To test the performance of dynamic update/synchronization, we experiment on vertex addition and deletion on the large *Synthetic graph*. To assure updates are indeed executed globally, 5 primary (complementary) partition sets are initially loaded and runs in parallel. In the experiment of vertex addition, we generate new vertices with respect to the degree distribution of the graph, which is a "power-law" distribution with $\gamma = 2.43$ (a.k.a *scaling parameter*, [2]). New edges are constructed according to preferential attachment. As to the experiment of vertex deletion, we randomly select vertices in the graph to delete. Figure 14 shows the average run time for each vertex addition/deletion operation by varying the number of vertices. It is observed that the addition and deletion operation per vertex can be accomplished in about 0.2ms and 0.4ms respectively and the time is almost constant with respect to the number of updated vertices.

8. CONCLUSIONS

We introduced an emerging data management problem in large-scale social and information networks. In order to process graph queries in parallel, these networks need to be partitioned and distributed across clusters. How to generate and manage partitions becomes an important issue. We illustrated that, for graph queries which have strong locality and skewed workload, static partition scheme does not work well. Thus, we proposed two partitioning techniques, *complementary partitioning* and *on-demand partitioning*. Based on these techniques, we introduced an architecture with a two-level partition structure, *primary* and *secondary partitions*, to handle graph queries with changing workload. The experiments demonstrated the developed system can effectively minimize inter-machine communication during distributed graph query processing. For future work, it is interesting to explore efficient RDF storage mechanisms and distributed metadata indexing solutions.

9. ACKNOWLEDGEMENTS

This research was sponsored in part by the U.S. National Science Foundation under grant IIS-0847925, ARO No. DFR 3A-8-447850-23002, and the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053 (NS-CTA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and

distribute reprints for Government purposes notwithstanding any copyright notation here on.

10. REFERENCES

[1] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[2] R. Albert and A.-L. Barabasi. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[3] M. Arenas and J. Pérez. Querying semantic web data with sparql. In *PODS*, 2011.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW*, pages 595–601, 2004.

[5] M. Bröcheler, A. Pugliese, V. P. Bucci, and V. S. Subrahmanian. COSI: Cloud oriented subgraph identification in massive social networks. In *ASONAM*, pages 248–255, 2010.

[6] J. Broekstra, A. Kampman, and F. V. Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, pages 54–68, 2002.

[7] P. Buneman, G. Cong, and W. Fan. Using partial evaluation in distributed query evaluation. In *VLDB*, pages 211–222, 2006.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[9] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[10] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, pages 48–57, 2010.

[11] A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, pages 301–312, 2011.

[12] J. Gilbert, G. Miller, and S.-H. Teng. Geometric mesh partitioning: implementation and experiments. *SIAM J. Sci. Comput.*, 19:2091–2110, 1998.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[14] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.

[15] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[16] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. of Supercomputing*, 1995.

[17] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. In *VLDB*, 2011.

[18] G. Jeh and J. Widom. Scaling personalized web search. In *WWW*, pages 271–279, 2003.

[19] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359 – 392, 1999.

[21] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA*, pages 13–21, 1992.

[22] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[23] J. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000.

[24] D. Kossmann. The state of the art in distributed query processing. *ACM Trans. Database Syst.*, 32(4):422–469, 2000.

[25] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[26] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental algorithms. In *SOCC*, pages 51–62, 2010.

[27] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[28] M. Najork. The scalable hyperlink store. In *Hypertext*, pages 89–98, 2009.

[29] M. Newman, A. L. Barabasi, and D. J. Watts. *The Structure and Dynamics of Networks*. Princeton University Press, 2006.

[30] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[31] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN*, pages 493–498, 1996.

[32] J.-M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.

[33] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A sparql performance benchmark. In *ICDE*, pages 222–233, 2009.

[34] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[35] D. Suciu. Distributed query evaluation on semistructured data. *ACM Trans. Database Syst.*, 27(1):1–62, 2002.

[36] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. of Parallel and Distributed Computing*, 47(2):102–108, 1997.

[37] K. Wilkinson and K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.

[38] D. R. Zerbino and E. Birney. Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Res.*, 18(5):821–829, 2008.

Appendix: The SP^{2}Bench Benchmark Queries

The queries used in the evaluation on the SP^{2}Bench Benchmark [33] are listed as follows.

*Q*2 Given an *inproceeding*, extract all the properties of the inproceeding, e.g., the title, the pages, the *authors*, the *proceeding* and the *reference list*.

*Q*4 Given a *journal*, select all distinct pairs of article author names for *authors* that have published in the *journal*.

*Q*6 Given a *proceeding* and a specific year, return the set of the *inproceedings* authored by persons that have not published in years before.

*Q*7 Given a *reference list*, return the titles of the papers in the list that have been cited at least once, but not by any paper that has not been cited itself.

*Q*8 Given an author, return the "*collaborative distance*" between the author and mathematician *Paul Erdös* (The distance is also known as *Erdös Number*).