LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

Jiewen Huang
Yale University
jiewen.huang@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

This paper introduces a dynamic graph partitioning algorithm, designed for large, constantly changing graphs. We propose a partitioning framework that adjusts on the fly as the graph structure changes. We also introduce a replication algorithm that is tightly integrated with the partitioning algorithm, which further reduces the number of edges cut by the partitioning algorithm. Even though the proposed approach is handicapped by only taking into consideration local parts of the graph when reassigning vertices, extensive evaluation shows that the proposed approach maintains a quality partitioning over time, which is comparable at any point in time to performing a full partitioning from scratch using a state-the-art static graph partitioning algorithm such as METIS. Furthermore, when vertex replication is turned on, edge-cut can improve by an order of magnitude.

1. INTRODUCTION

In recent years, large graphs are becoming increasingly prevalent. Such graph datasets are too large to manage on a single machine. A typical approach for handling data at this scale is to partition it across a cluster of commodity machines and run parallel algorithms in a distributed setting. Indeed, many new distributed graph database systems are emerging, including Pregel [22], Neo4j, Trinity [34], Horton [30], Pegasus [15], GraphBase [14], and GraphLab [21].

In this paper, we introduce data partitioning and replication algorithms for such distributed graph database systems. Our intention is not to create a new graph database system; rather our data partitioning and replication algorithms can be integrated with existing scalable graph database systems in order to improve the performance of the current implementation of their parallel query execution engines.

The most common approach for partitioning a large graph over a shared-nothing cluster of machines is to apply a hash function to each vertex of the graph, and store the vertex along with any edges emanating from that vertex on the machine assigned to that hash bucket. Unfortunately, hash partitioning graph data in this fashion can lead to suboptimal performance for graph algorithms whose access patterns involve traversing the graph along its edges. For example, when performing subgraph pattern matching, patterns are matched by successively traversing a graph along edges from partially matched parts of the graph. If the graph is partitioned in a way that nodes close to each other in the graph are physically stored as close to each other as possible, the network traffic of such graph algorithms can be significantly reduced.

In general, there are two goals that are desirable from a partitioning algorithm for workloads with traversal-oriented access patterns. First, if two vertices are connected by an edge, it is desirable for those two vertices to be stored on the same physical machine. If a partitioning algorithm assigns those vertices to two different machines, that edge is referred to as being "cut". Second, the size of the subgraph (in terms of the number of vertices and edges) stored on each partition should be approximately equal, so that graph algorithms can be parallelized equally across the cluster for maximum performance. These two goals are often in conflict. For example, it is easy to guarantee that no edge is ever cut if the entire graph is stored on a single machine, However, the partitioning would be extremely unbalanced, and graph algorithms would not be able to leverage the parallel resources in the cluster. On the other hand, hash partitioning usually gets near perfect balance of assignment of vertices and edges to nodes, but yields a high number of cut edges.

The dual goals of minimizing edge cut and maintaining balanced partitions is usually formulated as the k-balanced partitioning problem. k-partitioning has been studied extensively, and there exist many k-partitioning algorithms, both in the theoretical space, and some which have practical implementations. However, almost all of these k-partitioning algorithms run on static graphs — namely, the whole graph is known before the partitioning algorithm starts. The best known of these include METIS [17] and Chaco [12], which go through multiple levels of coarsening and refinement. Recent work includes lighter-weight algorithms that make several passes through the static graph and partition it without storing the entire graph in memory [35, 37, 25].

While getting a good initial partitioning of a static graph is certainly important, many modern graphs are dynamic, with new vertices and edges being added at high rates, and in some cases vertices and edges may be removed. Therefore, a good initial partitioning may degrade over time. It is inefficient to repeatedly run the static graph partitioning

Proceedings of the VLDB Endowment, Vol. 9, No. 7

algorithm to repartition the entire graph every time the partitioning starts to degrade a little. Instead, it is preferable to incrementally maintain a quality graph partitioning, dynamically adjusting as new vertices and edges are added to the graph. For this reason, there has been several recent research efforts in dynamic graph partitioning [33, 32, 31, 40, 39, 42].

Another important aspect of distributed systems is replication for fault tolerance by replicating data across several machines/nodes so that if one node fails, the data can still be processed by replica nodes. In general, the level of fault tolerance is specified by a minimum number of copies. If the entire subgraph stored on a particular node is replicated to an equivalent replica node, then replication can be considered completely independently from partitioning. However, more complicated replication schemes are possible where different parts of the subgraph stored on a node are replicated to different nodes depending on which nodes store subgraphs "closest" to that particular part of the subgraph. Replicating data in this way can significantly improve the edge-cut goals of partitioning, while maintaining the required fault tolerance guarantees.

One possible replication algorithm is to make replicas of all non-local neighbors for every vertex in the graph [27, 13] so that all accesses for neighbors are local. Unfortunately, many graphs contain "high degree" vertices (vertices associated with many edges), which end up getting replicated to most (if not all) nodes under such a replication algorithm. The effect is thus that some vertices get replicated far more than the minimum level required for fault tolerance, while other "low degree" vertices do not get replicated at all. A good replication algorithm needs to ensure a minimal replication for each vertex, while judiciously using any extra replication resources to replicate those vertices that will benefit the partitioning algorithm the most.

One problem with replication is keeping the replicas in sync. For graphs where the vertices contain attributes that are updated frequently, the update cost is multiplied by the number of replicas. Furthermore, many bulk-synchronous parallel (BSP) graph processing algorithms (such as well-known implementations of page rank, shortest paths, and bipartite matching [22]) work by updating ongoing calculations at each vertex upon each iteration of the algorithm. Enforcing replication of these running calculations at each iteration cancels out the edge-cut benefits of replication.

Previous approaches to dynamic graph partitioning were designed for graph applications where computations involve frequent passing of data between vertices. Therefore, they do not consider replication as a mechanism to reduce edge-cut, since the cost of keeping replicas in sync with each other during the computation is too high. In contrast, our work focuses on workloads that include read-only graph computations. While vertices and edges may be frequently added or deleted from the graph, and they may even occasionally be updated though explicit graph update operations, the processing operations over the graph fall into two categories — "read-only" and "non-read-only". Replication has the potential to improve the locality of read-only operations without hindering the performance of non-read-only operations that involve writing temporary data that can be deleted at the end of the computation (since the replicas can be ignored in such a scenario). Perhaps the most common read-only operation is sub-graph pattern matching and graph isomorphism — such operations are prevalent in SPARQL database systems where queries are expressed as a subgraph pattern matching operation over RDF data. However, read-only operations are also common for other types of graph systems. For example, triangle finding over social network data is an important tool for analyzing the properties of the network.

The primary contribution of our work is therefore the introduction of the first (to the best of our knowledge) dynamic graph partitioning algorithm that simultaneously incorporates replication alongside dynamic partitioning. We first propose a new, lightweight, on-the-fly graph partitioning framework that incorporates new edges instantly, and maintains a quality partitioning as the graph structure changes over time. Our approach borrows techniques and calculations from single-pass streaming approaches to (non-dynamic) graph partitioning. We leverage the flexibility of the streaming model to develop a vertex replication policy that dramatically improves the edge-cut characteristics of the partitioning algorithm while simultaneously maintaining fault tolerance guarantees. Since the reduction in edge-cut that is derived from replication is only applicable for certain graph operations, we also contribute a model that specifies the *effective edge-cut* of a replicated graph given a specific class of graph algorithm.

We also run an extensive evaluation of Leopard — an implementation of our dynamic partitioning and replication algorithm — on eleven graph data sets from various domains: Web graphs and social graphs and synthetic graphs. We find that even without replication, our stream-based dynamic partitioning approach produces an "edge-cut" similar to the edge-cut of the same graph if it had been repartitioned using a state-of-the-art static partitioning algorithm at that point in time. However, once replication is turned on, Leopard produces edge-cuts many factors smaller — in two cases transforming datasets with 80% edge-cut to 20% edge-cut.

2. BACKGROUND AND RELATED WORK

Even without considering dynamic graphs, static graph partitioning by itself is an NP-hard problem [11] — even the simplest two-way partitioning problem is NP-hard [10]. There are several linear programming-based solutions that obtain an $O(\log\ n)$ approximation for k-way partitioning [1, 8]. However, these approaches tend to be impractical. Various practical algorithms have been proposed that do not provide any performance guarantees, but prove efficient in practice. For small graphs, algorithms such as KL [19] and FM [9] are widely used. For large graphs, there are several multi-level schemes, such as METIS [17], Chaco [12] , PMRSB [3] and Scotch [26]. These schemes first go through several levels of coarsening to roughly cut the graph into small pieces, then refine the partitioning with KL [19] and/or FM [9] and finally project the pieces back to the finer graphs. These algorithms can be parallelized for improved performance, such as in ParMetis [18] and Pt-Scotch [5]. To handle billion-node graphs, Wang et al. [41] design a multi-level label propagation method on top of Trinity [34]. It follows the framework of coarsening and refinement, but replaces maximal match with a label propagation [29] method to reduce memory footprint.

Most of the research on partitioning of dynamic graphs focus on repartitioning the graph after a batch of changes are made to an original partitioning that cause the original partitioning to deteriorate [33]. In general, two approaches

are used. The first approach is scratch-map partitioning, which simply performs a complete partitioning again using a static partitioner [32]. The second (more commonly used) approach is diffusive partitioning. It consists of two steps: (1) a flow solution that decides how many vertices should be transferred between partitions using linear programming and (2) a multi-level diffusion scheme that decides which vertices should be transferred [31, 40]. These repartitioning schemes are heavyweight and tend to process a batch of changes instead of one change at a time. Furthermore, they tend to come from the high performance computing community, and do not consider replication alongside partitioning. In contrast, in this paper we focus on light-weight dynamic partitioning schemes that continually update the partitioning as new changes are streamed into the system, and tightly integrate replication with partitioning.

More recent works introduce light-weight partitioning algorithms for large-scale dynamic graphs. Vaquero et al. [39] propose a framework that makes use of a greedy algorithm to reassign a vertex to a partition with the most neighbors, but at the same time defers some vertex migration to ensure convergence. Xu et al. [42] propose methods for partitioning based on a historical log of active vertices in a graph processing system. Our work differs from theirs since we do not assume any kind of historical information of how the graph is processed. More importantly, the systems proposed by Vaquero and Xu et al. are designed for BSP graph processing systems that continuously update vertices in the graph with temporary data associated with running computations. Therefore, they do not consider replication as a mechanism for improving edge-cut, due to the overhead of keeping replicas updated with this temporary data. In contrast, a central contribution of our work is the consideration of read-only graph algorithms (such as subgraph pattern matching operations, triangle finding, and certain types of graph traversal operations) for which replication of data has the potential to reduce edge-cut and greatly improve the performance of these read-only operations. Thus the integration of partitioning and replication is the focus of our work, but not the other works cited above.

There have been several recent papers that show how treating replication as a first class citizen in graph database systems can greatly improve performance. Pujol et al. [27] use replication to ensure that neighbors of a vertex are always co-located in the same partition as the vertex. In other words, if a vertex's neighbors are not originally placed in the same partition as a vertex, they are replicated there. However, such a strong guarantee comes with significant overhead — high degree vertices get replicated a large number of times, and keeping the replicas consistent requires significant communication. Mondal and Deshpande fix this problem by defining a novel fairness requirement to guide replication that ensures that a fraction of neighbors (but not necessarily all neighbors) of a vertex be located in the same partition [23]. Furthermore, they use a clustering scheme to amortize the costs of making replication decisions. However, the main focus of the Mondal and Deshpande paper is replication, and their system is built on top of a hash partitioner. In contrast, our work focuses on reducing edge-cut through a combination of light-weight graph partitioning and replication. Partitioning and replication are built together as a single component in Leopard, and work in conjunction to increase locality of query processing.

Duong et al. show that another important benefit of replication is to alleviate load skew at query time [7]. Without replication, the partitions that store "popular nodes" get overwhelmed with requests for the values of those nodes. Replicating these popular nodes spreads out the resources available to serve these requests and reduces load skew. Similar to the work by Pujol et al. and Mondal and Deshpande, the replication scheme performed by Duong et al. also reduces edge-cut by using the number of neighbors in a local partition to guide the replication algorithm. The algorithm presented by Duong et al. is for the complete partitioning of a graph. This work does not discuss incremental repartitioning as the graph structure changes over time (this is explictly left by Duong et al. for future work). In contrast, our focus in this paper is on the dynamic repartitioning problem. The integration of replication considerations into dynamic repartitioning algorithms is non-trivial, and has not been addressed by the above-cited related work.

Leopard's light-weight dynamic graph partitioning algorithm builds on recent work on light-weight static partitioners. These algorithms partition a graph while performing a single pass through the data — for each vertex read from the input graph dataset, they immediately assign it to a partition without the knowledge of the vertices that will be read afterward. Because of the online nature of these algorithms, lightweight heuristics are often used to decide where to assign vertices, including the linear deterministic greedy (LDG) approach [35] and FENNEL [37]. LDG uses multiplicative weights to maintain a balanced partitioning and FENNEL leverages modularity maximization [4, 24] to deploy a greedy strategy for maintaining balanced partitions. Nishimura and Ugander [25] advanced this work by introducing several-pass partitioning and stratified partitioning. Similarly, Ugander and Backstrom introduce an iterative algorithm where each iteration does a full pass through the data-set and uses a linear, greedy, label propagation algorithm to simultaneously optimize for edge locality and partition balance [38]. However, the single-pass algorithms, along with the Nishimura et. al. and Ugander et. al. algorithms are designed to be run on demand, in order to occasionally (re)partition the entire graph as a whole. Furthermore, they do not consider replication. In contrast, our focus in this paper is on continuously repartitioning a graph in a lightweight fashion as it is dynamically modified. Furthermore, we explicity consider replication for fault tolerance as a tool for simultaneously improving partitioning.

3. VERTEX REASSIGNMENT

One key observation that serves as the intuition behind Leopard, is that the dynamic graph partitioning problem is similar to the one-pass partitioning problem [35, 37] described in Section 2. In many cases, what makes most dynamic graphs dynamic are new vertices and edges that are added. Therefore at any point in time, *a dynamic graph can be thought of as an intermediate state in the midst of a one-pass partitioning algorithm.* Consequently, the heuristics that have been used for one-pass algorithms can be leveraged by Leopard.

The intuition behind the one-pass streaming heuristics is the following: to achieve a low cut ratio, a new vertex should be assigned to the existing partition with most of its neighbors. However, at the same time, a large partition should be penalized to prevent it from becoming too large.

The FENNEL scoring heuristic is presented in Equation 1. P_i refers to the vertices in the ith partition. v refers to the vertex to be assigned and $N(v)$ refers to the set of neighbors of v. α and γ are parameters.

$$\underset{1 \leq i \leq k}{\text{argmax}}\{|N(v) \cap P_i| - \alpha \frac{\gamma}{2}(|P_i|)^{\gamma-1}\}, \qquad \text{(Equation 1)}$$

This heuristic takes a vertex v as the input, computes a score for each partition, and places v in the partition with the highest score. $|N(v) \cap P_i|$ is the number of neighbors of v in the partition. As the number of neighbors in a partition increases, the score of the partition increases. To ensure a balanced partitioning, it contains a penalty function based on the number of vertices, $|P_i|$, in the partition. As the number of vertices increases, the score decreases.

While one-pass partitioning algorithms are able to get a good initial partitioning at very low overhead, they fall short in three areas that are important for replicated, dynamic partitioning (1) a vertex is assigned only once and never moves, (2) they do not consider deletes, and (3) they do not consider replication. Therefore, we cannot use a simple adaptation of the one-pass streaming algorithms.

We first discuss the introduction of selective reassignment into streaming algorithms in this section. We defer the description of the full dynamic partitioning algorithm that includes a tight integration with replication to Section 4.

Example 1 illustrates a motivation for reassignment.

EXAMPLE 1. *Suppose there are two partitions P_1 and P_2. For the sake of simplicity of this example, we assume that the two partitions are roughly balanced at all times, so we only focus on the number of neighbors of a vertex to determine where it should be placed.*

After loading and placing the first 100 *edges into different partitions, vertex v has* 3 *neighbors in P_1 and* 2 *neighbors in P_2. Hence, P_1 is the better partition for v. However, after loading and placing another* 100 *edges, vertex v now has* 4 *neighbors in P_1 and* 10 *neighbors in P_2. The better partition for v is now P_2. Therefore, the optimal partition for a vertex changes over time as new edges are added (or deleted) — even though most of these new edges are added to a partition different than where v is located.*

Leopard therefore continuously revisits vertices and edges that have been assigned by the one-pass partitioning algorithm, and reassigns them if appropriate. Theoretically, all vertices could be examined for reassignment every time an edge is added (or deleted). However, this is clearly impractical. Instead Leopard uses a more strategic approach.

When an edge (v_1, v_2) is added or deleted, most vertices are minimally affected. Only those vertices in the graph near the edge are likely to be influenced enough to potentially justify a move. If v_1 and v_2 are located on different partitions, they become good candidates for reassignment because they now have one additional (or less) neighbor on a different partition. In particular, v_1 should perhaps be moved to v_2's partition, or vice versa. This may cause a ripple effect among neighbors of a reassigned vertex — once a vertex moves to a new machine, its neighbors may also be better off being moved to a new machine.

Therefore, when a new edge (v_1, v_2) is added, v_1 and v_2 are chosen as the initial candidates for examination of potential reassignment. If either one of them are reassigned, then the immediate neighbors of the moved vertex are added to the candidate set. This candidate set thus contains the set of vertices immediately impacted by the recent graph activity. A subset of the vertices in this candidate set are examined for potential reassignment. This examination process involves reapplying the score function (e.g., Equation 1) to these candidates. The algorithm for choosing the particular subset of candidates that will be re-scored and potentially reassigned is given in the following section.

3.1 Timing of Reassignment Examination

After examination candidates are chosen, Leopard must decide whether examination should be performed on those candidates. As described above, examining a vertex v for reassignment involves calculating a score (e.g., Equation 1) for the placement of v on every partition in the cluster. This scoring process includes a calculation based on the number of neighbors of v in each partition, which requires information to be accessed regarding where v's neighbors are located. If the vertex location lookup table is small enough to be hosted on one machine, then the cost of the lookup is small. However, for massive graphs, the lookup table is spread across the cluster, and this cost can be significant (Leopard uses a distributed consistent hash table [16] that maps vertex identifiers to their respective partitions). In general, examination comes with a non-negligible cost, and should not be performed when there is little probability of the examination resulting in reassignment.

As a vertex's number of neighbors increases, the influence of a new neighbor decreases. For example, in one experiment that we ran in Section 6, a vertex with 10 neighbors is reassigned to a new partition upon receiving an 11th neighbor 13% of the time, while a vertex with 40 neighbors is reassigned to a new partition upon receiving a 41st neighbor only 2.5% of the time. Therefore, it is less cost-efficient to pay the reassignment examination cost for a vertex with a large number of existing neighbors, since it is likely that this examination work will be wasted, and the vertex will remain in its current partition.

Leopard therefore occasionally avoids the reassignment examination computation of a candidate vertex v if v already has many neighbors in the graph. Nonetheless, Leopard ensures that even vertices with a large number of neighbors get examined periodically. The pseudocode for the logic on when to skip examination is shown in Figure 1. Given a vertex v and a threshold t, the function skips the computation of the reassignment scores if the ratio of skipped computation to the total neighbors of v is less than t. This has the effect of making the probability of performing reassignment examination after adding a new neighbor to v proportional to $(1-t)/(t)(v.neighbors)$.

Upon breaking up this formula into its two components, it can be seen that there are two factors that determine whether an examination should be performed: $1/(v.neighbors)$ and $(1-t)/t$. This means that as the number of neighbors increases, the probability of examination for reassignment goes down. At the same time, as t approaches 0, examination will always occur, while as t approaches 1, examination will never occur.

There is clearly a tradeoff involved in setting t. A larger t skips more examinations and reduces the partitioning quality, but improves the speed of the algorithm. Section 6 revisits this tradeoff experimentally (see Figures 10 and 11).

```
Function ToExamineOrNot(v, threshold)
  if an edge is added then
    v.neighbors++
  else if an edge is deleted then
    v.neighbors−−
  end if
  if (v.skippedComp + 1) ÷ v.neighbors ≥ threshold then
      compute the reassignment scores for v
      v.skippedComp = 0
  else
      v.skippedComp++
  end if
```

Figure 1: Algorithm for deciding when to examine a vertex for reassignment.

4. LEOPARD PARTITIONING

We now present the complete Leopard partitioning algorithm that tightly integrates replication with dynamic partitioning.

Replication serves two purposes in Leopard: providing fault tolerance and improving access locality. Fault tolerance is achieved by replicating vertices to different machines. By having n copies of each vertex, every vertex is still accessible even if the system loses $n-1$ machines. Some graph database systems are built on top of distributed file systems such as HDFS that provide their own replication mechanisms for fault tolerance. Such mechanisms are redundant with Leopard's replication and should be avoided if possible when used in conjunction with Leopard (for example, HDFS has a configuration parameter that removes replication).

Better access locality improves performance. Even if an edge (v_1, v_2) is cut (meaning that v_1 and v_2 are in separate partitions, which usually results in them being stored on separate machines), if a copy of v_2 is stored on v_1's partition (or vice versa), computation involving v_1 and v_2 may still proceed locally without waiting for network communication.

Most systems treat replication for access locality and fault tolerance separately [7, 13, 27]. For example, a partitioner will create an initial partitioning, and certain important vertices are selectively replicated to multiple partitions in order to reduce the edge-cut. Afterwards, entire partitions (or shards of partitions) are replicated to other machines for fault tolerance. This has the effect of "double-replicating" these important vertices. This may not be the most efficient use of resources. A better edge-cut may have been achievable had additional important vertices been replicated during the first phase instead of double-replicating the original set of important vertices. Leopard thus takes the approach of simultaneously considering access locality and fault tolerance in order to avoid this waste of resources. It is important to note, however, that recovery from failures are more complicated when multiple shards or partitions need to be accessed in order to recover the data on the failed machine. Although novel in the graph database space, this approach of trading off recovery complexity for improved resource utilization has been performed before in the context of column-store database systems. Different combinations of columns (in potentially different sort orders) are stored on different machines, while ensuring that at least k copies of each column are located across the cluster [36].

The primary downside to replication is the cost of keeping replicas in sync with each other. If an update to the data stored inside a vertex must be performed, this update must be propagated to all replicas, and this must happen atomically if replicas are not allowed to not temporarily diverge for a particular application. In this paper, we classify updates into two categories: permanent updates to fundamental attributes of vertices or edges in a graph (e.g. changing the name of a user in a social network), and temporary updates to vertices or edges that occur over the course of a graph computation. In this section, we assume that the former type of updates happen far less frequently than reads of these vertices or the addition or removal of vertices and edges. However, several solutions have been proposed in the research community for how to handle replication when this assumption does not hold [6, 28]. We make no assumptions about the latter type of updates (temporary updates) and discuss them in more detail below.

4.1 Replication, Access Locality and Cut

An access from a vertex v to a neighbor u is local if v and u are placed on the same machine. Otherwise it is remote. When vertices are not replicated, this is directly related to whether edge (v, u) is cut, and edge cut ratio can be used to predict access locality. However, the definition of cut ratio is more complicated when considering vertex replication.

Figure 2 illustrates this complexity for a sample graph containing vertices in triangle. In the case without replication (a), all three edges are cut. In the case with replication (b), A, B and C are replicated to partitions 3, 1 and 2, respectively. It could be argued that the edge (A, B) is no longer cut, since partition 1 now contains a copy of both A and B. By the same logic, the edge (A, C) would no longer be cut since partition 3 now contains both vertices. However, if a query wanted to access A, B, and C, this query cannot be performed locally. In other words, even though the edge (A, B) is not cut, and the edge (A, C) is not cut, we cannot transitively deduce that A, B, and C are in the same partition. In contrast, in a system without replication, such a transitive property would hold.

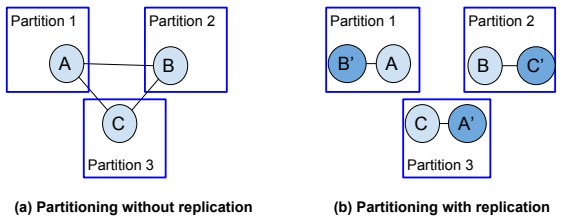

Figure 2: Partitions without and with replication. A', B' and C' are replicas.

The fundamental problem is that replication adds a direction to an edge. When B is replicated to A's partition, it allows B to be accessed from the partition with the master copy of A. However, it does not allow A to be accessed from the partition with the master copy of B. Therefore half of all co-accesses of A and B will be non-local (those that initiate from B's master copy) and half are local (those that initiate from A's master copy). In other words, vertex replication repairs the "cut" edges between the replicated vertex and all vertices on the partition onto which it is replicated, but these edges are only "half"-repaired — only half of all accesses to that edge will be local in practice.

Furthermore, in some cases Leopard chooses to let the replicas diverge. In particular, temporary data that is associated with every vertex during an iterative graph computation (that is updated with each iteration of the computation) are not propagated to replicas. The reason for this is that it only "pays off" to spend the cost of sending data over the network to keep replicas in sync if each replica will be accessed multiple times before the next update. For iterative graph computation where updates occur in every iteration, this multi-access requirement is not met. Therefore, Leopard does not propagate this temporary data to replicas during the computation, and thus replica nodes cannot be used during the computation (since they contain stale values). In such a scenario, the *effective edge cut* with replication is identical to the edge cut without replication.

In order to accurately compare the query access locality of an algorithm over a replicated, partitioned graph against the locality of the same algorithm over an unreplicated graph, we focus on *effective edge cut*, which prevents an overstatement of the benefit of replication on edge-cut. In order to precisely define *effective edge cut*, we classify all graph operations as either "read-only" and "non-read-only". Read-only operations do not write data to vertices or edges of the graph during the operation. For example, sub-graph pattern matching, triangle finding, and certain types of graph traversal operations are typically read-only. In contrast, non-read-only operations, such as the iterative BSP algorithms discussed above, potentially write data to the graph. The *effective edge cut* is defined relative to the classification of a particular operation being performed on a graph. Given an operation, O, and an edge (u, v), Figure 3 provides a value for the effective cut for that edge. When calculating the edge-cut for the entire graph, edges for which `DefineEffectiveCut` returns "HALF CUT" have only half of the impact relative to fully cut edges on the final edge-cut value. This accounts for the uni-directional benefit of replicating a vertex to a new partition.

Figure 4 shows examples corresponding to the four cases presented in the edge cut definition for read-only operations.

```
Function DefineEffectiveCut(v, u, O)
  // p(v) and s(v) denote the primary copy
  // and a secondary copy of v, respectively.
  // O is the operation being performed on the graph.
  if p(v) and p(u) are on the same partition then
      return NOT CUT
  else if p(v) and s(u) are on the same partition &
          p(u) and s(v) are on the same partition then
      if O is read-only
        return NOT CUT
      else
        return CUT
  else if p(v) and s(u) are on the same partition |
          p(u) and s(v) are on the same partition then
      if O is read-only
        return HALF CUT
      else
        return CUT
  else
      return CUT
  end if
```

Figure 3: A definition of effective edge cut in the presence of vertex replication.

v		u		Edge (v, u)
primary	second.	primary	second.	
1	2,3	1	4,5	NO CUT
1	2,3	2	1,3	NO CUT
1	2,3	2	3,4	HALF CUT
1	3,4	2	3,4	CUT

Figure 4: Examples of location of primary and secondary copies and the corresponding cut for a read-only operation. A value of 2,3 in the secondaries column means that the secondary copies are placed on partitions 2 and 3.

4.2 Minimum-Average Replication

Leopard uses a replication scheme called MAR, an acronym of **M**inimum-**A**verage **R**eplication. As the name suggests, the scheme takes two parameters: the minimum and average number of copies of vertices. The minimum number of copies ensures fault tolerance and any additional copies beyond the minimum provides additional access locality. In general, the average number of copies specified by the second parameter must be larger than or equal to the first parameter. Figure 5 shows an example graph with MAR.

Given a vertex, the MAR algorithm decides how many copies of it should be created and in which partitions they should be placed. Leopard uses a two step approach. First, a modified version of the vertex assignment scoring algorithm is run. Second, the scores generated from the first step are ranked along with recent scores from other vertices. Vertices with a high rank relative to recent scores get replicated at a higher rate than vertices with relatively low scores.

4.2.1 Vertex assignment scoring with replication

Because of replication, the assignment score functions need to be modified to accommodate the presence of secondary copies of vertices and to remain consistent with the new edge cut definition given in Section 4.1. In particular the score function (see e.g., Equation 1) requires the list of all neighbors of the vertex being scored on each partition. Given a vertex v that is currently being scored, we make the definition of whether a vertex u is a neighbor of v in partition P dependant on whether we are currently scoring the primary or secondary copy of v:

DEFINITION 1. *Consider a graph $G = (V, E)$. $p(v)$ and $s(v)$ denote the primary copy and a secondary copy of v, respectively. u is a neighbor of v on partition P if $(v,u) \in E$ and also one the following holds:*
(1) v is a primary copy and $p(u) \in P$
(2) v is a primary copy and $s(u) \in P$
(3) v is a secondary copy and $p(u) \in P$

Given that this definition is dependent on whether the primary or secondary copy of v is being scored, Leopard computes two scores for each partition, P: one for the primary copy of v and one for secondary copies. The primary copy will be placed in the partition with the highest primary copy score, and secondary copies are placed according to their scores using the algorithm presented in Section 4.2.2.

4.2.2 Ranking scores for secondary copies

After computing the score for a secondary copy of vertex v for each candidate partition, these scores are sorted from highest to lowest. Since there is a minimum requirement of

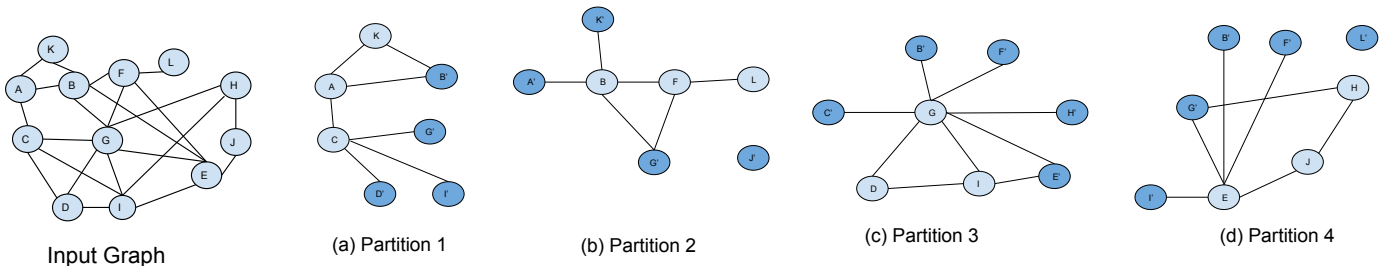

Figure 5: Example of Minimum-Average Replication of an input graph, where the minimum = 2 and average = 2.5. The primary copy of each vertex is shown in light blue and their secondary copies are highlighted in a darker blue. 8 vertices have 2 copies, 2 have 3 copies, and 2 (the best-connected vertices) have 4 copies. The copy of J in partition 2 does not improve access locality, but is required for fault tolerance.

Function `ChoosePartitions`(v, min, $average$)

choose a partition for the primary copy of v:
  run the scoring algorithm for primary copy location
  $P_{primary}$ = the partition with the highest score
  Place primary copy of v on $P_{primary}$

choose partitions for the secondary copies of v:
  run the scoring algorithm for secondary location (
    excluding $P_{primary}$) and get a set of scores A
  sort A from high to low
  P_{min} = $(min - 1)$ partitions with the highest scores
  B = all recently computed secondary copy scores
  $C = B \bigcup A$
  sort C from high to low
  /* k (see below) is the number of partitions */
  $P_{average}$ = partitions whose score appear in the
    top $\frac{average-1}{k-1}$ of all scores
  $P_{secondary} = P_{min} \bigcup P_{average}$
  Place secondary copies of v on partitions in $P_{secondary}$

Figure 6: An algorithm to choose partitions for the primary and secondary copies for a vertex.

copies, M, necessary for fault tolerance, secondary copies are immediately placed in the partitions corresponding to the top $(M-1)$ scores. However, the MAR parameter corresponding to the average number of copies, A, is usually higher than M, so additional copies may be created. To decide how many copies to make, Leopard compares the scores for v with the s most recent scores for other vertices. Commensurate with the extent that the scores for v are higher or lower than the average scores for the s most recent vertices, Leopard makes the number of copies of v higher or lower than A. The specific details of how many copies are created are presented in the algorithm in Figure 6.

There are two reasons why the comparison of v's scores are only made with the most recently computed scores of s vertices, instead of all scores. First, for a big graph, the number of all computed scores is large. It is space consuming to store them and time consuming to rank them. Second, as the graph grows larger, the scores tend to rise. Therefore, the scores computed at the initial stages are not comparable to the scores computed at later stages. The parameter s, representing the sliding window of score comparisons, is a customizable parameter. In practice, s can be small (on the order of 100 scores), as the function of this window is only to get a statistical sample of recent scores. Thus, this window of scores takes negligible space and easily fits in memory.

The following example illustrates how MAR works.

EXAMPLE 2. *Assume parameters of M (minimum) = 2 and A (average) = 3 for MAR and the number of partitions is 5. Also assume the sliding window s has a size of 24.*

When running the vertex assignment scoring algorithm for the primary copy of a vertex v, partitions 1 through 5 get scores of 0.15, 0.25, 0.35, 0.45 and 0.55, respectively. Since partition 5 has the highest score, it is chosen as the partition for the primary copy of v.

After running the vertex assignment scoring algorithm for the secondary copies of v, partitions 1 through 4 get scores of 0.1, 0.2, 0.3 and 0.4, respectively (partition 5 is excluded because it already has the primary copy). Since M is 2 and partition 4 has the highest score, it is immediately chosen as the location for a secondary copy (to meet the minimum copy requirement). Including partition 5, there are now two partitions with copies of v.

The four scores for the secondary copies (i.e., 0.1, 0.2, 0.3 and 0.4) are then combined with the 24 recent scores in the sliding window s and all 28 scores are then sorted. If any of v's three other secondary scores are in the top $\frac{average-1}{k-1}$ $= \frac{3-1}{5-1} = 50\%$ of all scores, then additional copies of v are made for those partitions.

5. ADDING AND REMOVING PARTITIONS

In some circumstances, the number of partitions across which the graph data is partitioned must change on the fly. In these cases, Leopard needs to either spread out data to include new partitions or concentrate the data to fewer partitions. Decreasing the number of partitions is relatively straightforward in Leopard. For every vertex placed on the partitions to be removed, we run the vertex reassignment scoring algorithm and assign it to the best partition accordingly. We will therefore now focus on adding new partitions.

A simple approach to adding a new partition would be to either use a static partitioner to repartition the entire graph over the increased number of partitions, or to reload the entire graph in Leopard with a larger value for P (the number of partitions). Unfortunately, performing a scan through the entire graph to repartition the data is expensive, and is particularly poor timing if the reason why partitions are being added is because machines are being added to accommodate a currently heavy workload.

Graph	|V|	|E|	Density	Clustering Coef.	Diameter [20]	Type
Wiki-Vote (WV)	7,115	100,762	$3.9 * 10^{-3}$	0.1409	3.8	Social
Astroph	18,771	198,050	$1.1 * 10^{-3}$	0.6306	5.0	Citation
Enron	36,692	183,831	$2.7 * 10^{-4}$	0.4970	4.8	Email
Slashdot (SD)	77,360	469,180	$1.6 * 10^{-4}$	0.0555	4.7	Social
NotreDame (ND)	325,729	1,090,108	$2.1 * 10^{-5}$	0.2346	9.4	Web
Stanford	281,903	1,992,636	$5.0 * 10^{-5}$	0.5976	9.7	Web
BerkStan (BS)	685,230	6,649,470	$2.8 * 10^{-5}$	0.5967	9.9	Web
Google	875,713	4,322,051	$1.1 * 10^{-5}$	0.5143	8.1	Web
LiveJournal (LJ)	4,846,609	42,851,237	$3.7 * 10^{-6}$	0.2742	6.5	Social
Orkut	3,072,441	117,185,083	$2.5 * 10^{-5}$	0.1666	4.8	Social
BaraBasi-Albert graph (BA)	15,000,000	1,800,000,000	$1.6 * 10^{-5}$	0.2195	4.6	Synthetic
Twitter	41,652,230	1,468,365,182	$1.7 * 10^{-6}$	0.1734	4.8	Social
Friendster (FS)	65,608,366	1,806,067,135	$8.4 * 10^{-7}$	0.1623	5.8	Social

Figure 7: Statistics of the graphs used in the experiments. Diameter is reported at 90th-percentile to eliminate outliers.

Instead, Leopard first selects a group of vertices as seeds for a new partition, and then runs the vertex reassignment scoring algorithm on their neighbors, the neighbors of their neighbors, the neighbors of the neighbors of their neighbors, and so on until all partitions are roughly balanced. Leopard proceeds in this order since the neighbors of the seed vertices that were moved to the new partition are the most likely to be impacted by the change and potentially also move to the new partition. This results in a requirement to compute the reassignment scores for only a small, local part of the graph, thereby keeping the overhead of adding a new partition small.

There are several options for the initial seeds for the new partitions.

- Randomly selected vertices from all partitions.
- High-degree vertices from all partitions.
- Randomly selected vertices from the largest partitions (to the extent that there is not a perfect balance of partition size).

We experimentally evaluate these approaches in Section 6.4.

6. EVALUATION

6.1 Experimental Setup

Our experiments were conducted on 4th Generation Intel Core i5 and 16GB memory with Ubuntu 14.04.1 LTS.

6.1.1 Graph datasets

We experiment with several real-world graphs whose sizes are listed in Figure 7. They are collected from different domains, including social graphs, collaboration graphs, Web graphs and email graphs. The largest graph is the Friendster social graph with 66 million vertices and 1.8 billion edges.

We also experiment with a synthetic graph model, the well-known BaraBasi-Albert (BA) model [2]. It is an algorithm for generating random power-law graphs with preferential attachment. For the parameters to this model, we used $n = 15{,}000{,}000$, $m = 12$ and $k = 15$.

We transform all of these graphs into dynamic graphs by generating a random order of all edges, and streaming edge insertion requests in this order. Although this method for generating dynamic graphs does not result in deletions in the dynamic workload, we have also run experiments that include deletions in the workload, and observed identical results due to the parallel way that Leopard handles additions and deletions of edges.

6.1.2 Comparison Points

We consider the following five partitioning approaches as comparison points.

1. **Leopard.** Although Leopard supports any vertex assignment scoring function, for these experiments we use the same scoring function used by Tsourakakis et al. (FENNEL) [37]. As suggested by Tsourakakis et al., the FENNEL parameters we use throughout the evaluation is $\gamma = 1.5$ and $\alpha = \sqrt{k}\frac{|E|}{|V|^{1.5}}$. The sliding window we use for the score ranking mechanism described in Section 4.2.2 contains up to 30 recent vertices.

2. **One-pass FENNEL partitioning.** This comparison point is a traditional one-pass streaming algorithm with the FENNEL scoring function (with the same parameters as above). Like all traditional streaming algorithms, after a vertex has been placed in a partition, it never moves. This comparison point is used to evaluate the benefit of vertex reassignment in Leopard.

3. **METIS [17].** METIS is a state-of-the-art, widely-used static graph partitioning algorithm. It does not handle dynamic graphs, so we do not stream update requests to METIS the same way we stream them to Leopard and FENNEL. Instead, we give METIS the final graph, after all update requests have been made, and perform a static partitioning of this final graph. Obviously, having a global view of the final graph at the time of partitioning is a big advantage. Therefore, METIS is used as an approximate upper bound for partitioning quality. The goal for Leopard is to get a similar final partitioning as METIS, despite not having a global view of the final graph while partitioning is taking place.

4. **ParMETIS [31].** ParMETIS is a state-of-the-art parallel graph partitioning algorithm. It is designed for a different type of dynamic graph partitioning than Leopard. In particular, ParMETIS is designed for bulk repartitioning operations after many vertices and edges have been added to the graph. This makes comparison to Leopard non-trivial. Leopard continuously maintains a good partitioning of a graph, while ParMETIS allows a partitioning to deteriorate and then occasionally fixes it with its batch repartitioner. We thus report two separate sets of ParMETIS results — shortly before and after the batch repartitioner has been run. At any point in

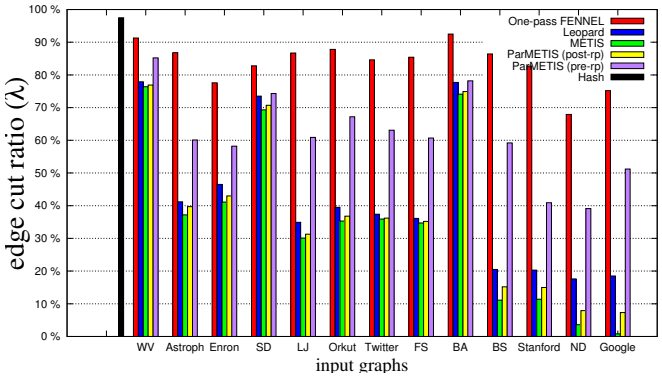

Figure 8: Edge cut experiment. Cut ratio for hash partitioning is independent of the data set, and so it is displayed just once on the left-hand side.

time, ParMETIS will have a partitioning quality in between these two points. In order to present ParMETIS in a good light, we use FENNEL to maintain a reasonable quality partitioning in between repartitioning operations as new vertices and edges are added to the graph (instead of randomly assigning new vertices and edges to partitions until the next iteration of the repartitioner).

5. **Hash Partitioning.** Most modern scalable graph database systems still use a hash function to determine vertex location. Therefore, we compare against a hash partitioner, even though it is known to produce many cut edges.

6.1.3 Metrics

We evaluate the partitioning approaches using two metrics: the edge cut ratio λ and load imbalance ρ, which are defined as

$$\lambda = \frac{\text{the number of edges cut}}{\text{the total number of edges}}$$

(For replicated data, a cut edge is defined as in Figure 3)

$$\rho = \frac{\text{the maximum number of vertices in a partition}}{\text{the average number of vertices in a partition}}.$$

6.2 Dynamic Partitioning

6.2.1 Comparison of Systems

In this section, we evaluate partitioning quality in the absence of replication. We partition the eleven real-world graphs into forty partitions using the five partitioning approaches described above and report the results in Figure 8. The figure only presents the edge-cut ratio, since ρ (balance) is approximately the same for all approaches (it has a value of 1.00 for hash partitioning and varies between 1.02 to 1.03 for the other partitioners).

As expected, hash partitioning performs very poorly, since the hash partitioner makes no attempt to place neighboring vertices on the same machine. 39/40 (which is 97.5%) of edges are cut, since with 40 partitions, there is a 1 in 40 probability that a vertex happens to end up on the same partition as its neighbor.

The "One-pass FENNEL" partitioner also performs poorly, since the structure of the graph changes as new edges are added, yet the algorithm is unable to adjust accordingly. When comparing this partitioner to Leopard, the importance of dynamic reassignment of graph data becomes clear. On the other hand, Figure 8 does not present the computational costs of partitioning. We found that the FENNEL partitioner completes a factor of 44.6 times faster than Leopard. So while Leopard's partitioning is much better, it comes at significant computational cost. However, this experiment did not use the skipping optimization described in Section 3.1. In further experiments described below, we will find that this factor of 44.6 computational difference can be reduced to a factor of 2 with only small changes in cut ratio.

ParMETIS either performs poorly or well depending on how recently the batch repartitioner has been run. For this experiment, the batch repartitioner ran after loading one fourth, one half, and three fourths of the vertices and edges of the graph, and a fourth time at the end, after the entire graph dataset had been loaded. We present two results for ParMETIS - shortly before this final batch repartitioning is performed (labeled ParMETIS-pre-rp), and directly after the final repartitioning is performed (labeled ParMETIS-post-rp). ParMETIS-pre-rp is the worst case scenario for ParMETIS — it has been the longest possible time since the last repartitioning. In this period of time, vertices and edges were added to the graph according to the FENNEL heuristic. ParMETIS-post-rp is the best case scenario for ParMETIS — it has just done a global repartitioning of the graph. Although ParMETIS' repartitioning algorithm is lighter-weight than METIS' "from-scratch" partitioning algorithm, its edge-cut after this repartitioning operation is close to METIS.

Surprisingly, Leopard is able to achieve a partitioning very close to METIS and ParMETIS' best case scenario (post-rp), despite their advantage of having a global view of the final graph when running their partitioning algorithms. Since we use METIS as approximate upper bounds on partitioning quality, it is clear that Leopard is able to maintain a high quality partitioning as edges are added over time.

Although Leopard achieves a good partitioning for all the graphs, the quality of its partitioning relative to METIS is poorest for the Web graphs. An analysis of the statistics of the graphs from Figure 7 shows that the Web graphs have large diameters. Graphs with large diameters tend to be easier to partition. Indeed, a static partitioner with global knowledge is able to cut less than 1 in 10 edges in the Web graphs we used in our experiments. However, without this global knowledge, Leopard enters into local optima which results in more edges being cut.

These local optima are caused by the balance heuristics of the streaming algorithm. Leopard's scoring formula is designed to keep the size of each partition approximately the same (as reported above, Leopard achieves a ρ-balance of between 1.02 to 1.03). For graphs that are hard to partition, such as the power law graph we generated using the BaraBasi-Albert model, there are many vertices that no matter where they are placed, they will result in many edges being cut. Since the exact placement of these "problematic" vertices will not have a large effect on the resulting quality of the partitioning, they can be placed in the partition that will most help to keep ρ low. In contrast, for partitionable graphs, there are fewer of such "ρ-fixing" vertices, and occasionally vertices are placed on a partition with much fewer neighbors in order to avoid imbalance. This results in other vertices from the same partition to follow the original vertex

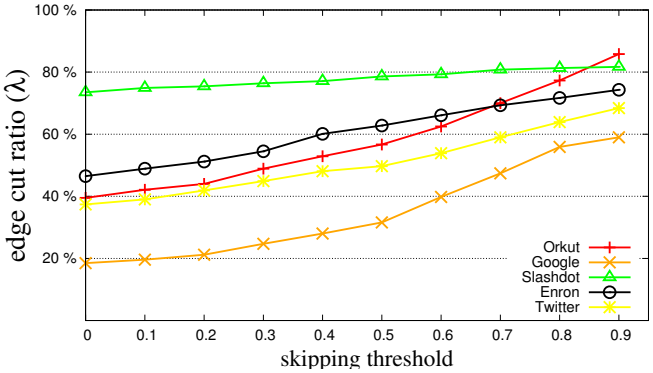

Figure 9: Effects of skipping examination on edge cut ratio.

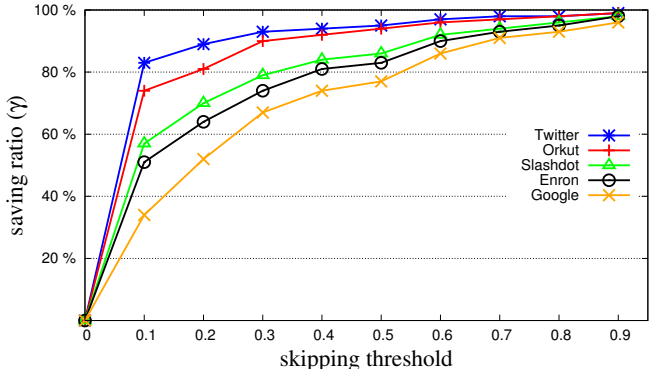

Figure 10: Computation savings vs. skipping threshold.

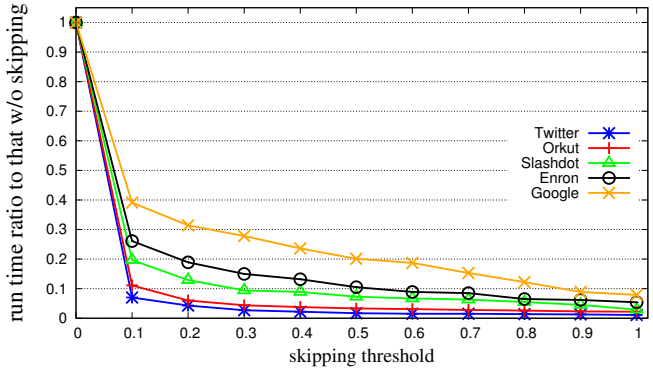

Figure 11: Run time improvement vs. skipping threshold.

to the different partition. The end result is that sets of vertices that should clearly be part of one partition occasionally end up on two or three different partitions.

This implies that Leopard should keep track of the neighbor locality and balance components of Equation 1 separately. When there is frequent disparity between these components, Leopard should occasionally run a static partitioner as a background process, to readjust the partitioning with a global view, in order to escape these local optima.

Although this experiment involves only adding new edges, we also ran an experiment where edges are both dynamically added and deleted. We found that the inclusion of edge deletions in the experiment did not make a significant difference relative to the workload with only addition of edges, since both addition and removal of edges are treated as "relevant" events that cause reassignment to be considered. We do not present the results in more detail here due to lack of space.

6.2.2 *Skipping*

In Section 3.1 we described a shortcut technique, where certain vertices with many existing neighbors (and are therefore unlikely to be reassigned) are not examined for reassignment, even if a new edge adjacent to that vertex is added. In our previous experiment, this shortcut was not used. We now turn it on, and look more closely at how skipping examination of vertices affects the edge cut ratio.

The results are shown in Figure 9. As expected, as the the skipping threshold increases (i.e., reassignment examination is skipped more often), the quality of the cut suffers. The lower the original cut ratio, the more damage skipping vertex examination can have on cut ratio. The Slashdot graph has close to a 80% ratio without skipping and the 0.9 skipping threshold only increases the cut ratio to slightly above 80%. On the other hand, Google Web graph's cut ratio dramatically jumps from 20% to 60%. However, for all graphs, skipping thresholds below 0.2 have a much smaller effect on cut ratio than higher thresholds.

Despite deteriorating the edge cut ratio, skipping vertex examination can save resources by only forcing the system to spend time examining vertices for reassignment if they are likely to actually be reassigned as a result of the examination. We define the savings as a fraction of the original number of examined vertices, γ, as:

$$\gamma = \frac{\text{number of vertices skipped for examination}}{\text{number of vertices examined at threshold 0}}.$$

A higher (lower) γ indicates higher (lower) savings of computation resources related to reassignment examination.

Figure 10 shows how changing the skipping threshold leads to reassignment examination savings. For dense graphs such as Orkut and Twitter (which have over 70 edges per vertex), a skipping threshold as low as 0.1 can result in an order of magnitude reduction in the reassignment examination work that must be performed. For sparser graphs, the benefits of skipping vertex examination are less significant.

Figure 11 shows how this reassignment examination savings translates to actual performance savings. The figure shows the end-to-end run-time as a ratio to that without skipping of the partitioning algorithm for the same experiment that was performed in Section 6.2.1. When comparing the extreme left- and right-hand sides of the graph, the cost of too much vertex reassignment is evident. The extreme right-hand side of the graph (where the skipping ratio is 1), results in there never being any reassignment ever. Leopard simply loads the vertices and edges one by one, and never moves a vertex from its initial placement. The extreme left-hand side of the graph corresponds to the full reassignment examination policy without any skipping. It is consistent with Figure 10 that skipping is more beneficial for denser graphs. We will use Orkut as an example to illustrate the benefit of skipping. The total time to pass through the Orkut graph without any reassignment is 215 seconds. The total time it takes to do the pass through the data to examine every possible reassignment jumps to 9584 seconds. This shows the significant cost of dynamic reassignment and reexamination if left unchecked. However, a skipping threshold of 0.2 reduces the run time by over a factor of 20 to 571 seconds. Note from Figure 9 that most of the

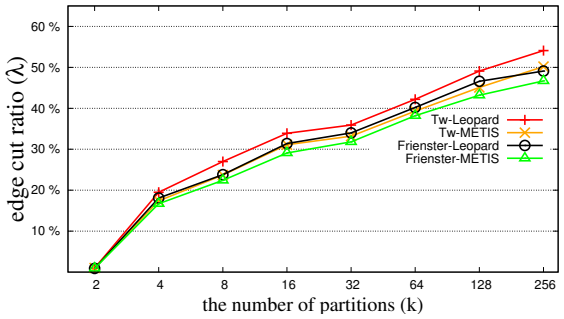

Figure 12: Effect of number of partitions on edge-cut on the Twitter graph

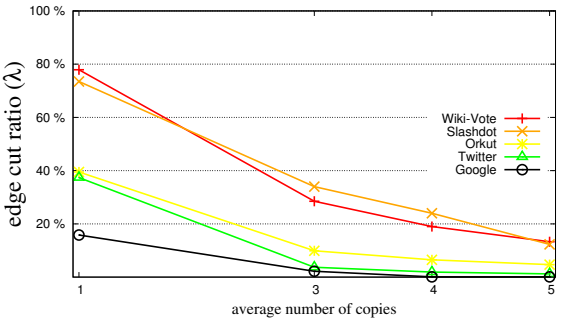

Figure 13: Effect of replication on edge-cut.

edge-cut benefits of Leopard remain with a skipping ratio of 0.2. Thus, while the skipping ratio parameter clearly leads to an edge-cut vs. run-time tradeoff, it is possible to get most of the edge-cut benefits without an overly-significant performance cost with reasonably small skipping ratios.

In summary, for resource constrained environments, a threshold between 0.1 and 0.2 produces a small decrease in partitioning quality, while significantly improving the efficiency of the reassignment process. Denser graphs can use smaller thresholds, since most of the efficiency benefits of the skipping optimization are achieved at thresholds below 0.1.

6.2.3 *Scalability in the Number of Partitions*

We now investigate how the cut quality changes as the number of partitions varies. For this experiment, we varied the number of partitions from 2 partitions to 256 partitions for the two largest graphs in our datasets — the Twitter and Friendster graphs. The results are shown in Figure 12. Both Leopard and METIS's partitioning quality gets steadily worse as the number of partitions increases. This is because of the fundamental challenge of keeping the partitions in balance. With more partitions, there are fewer vertices per partition. As clusters of connected parts of the graph exceed the size of a partition, they have to be split across multiple partitions. The denser the cluster, the more edges are cut by splitting it.

Although the quality of the cut ratio of both Leopard and METIS gets worse as the number of partitions increases, the *relative difference* between Leopard and METIS remains close to constant. This indicates that they have similar partition scalability.

6.3 Leopard with Replication

We now explore the properties of the complete Leopard implementation with replication being integrated with partitioning. Since Leopard is the first system (to the best of our knowledge) that integrates replication with dynamic partitioning, it is not fair to compare Leopard with replication to the comparison points we used above, which do not support replication as a mechanism to improve edge cut. Thus, in order to understand the benefits of incorporating replication into the partitioning algorithm, we compare Leopard with replication to Leopard without replication. However, since we run on the same datasets as used above, the reader can indirectly compare these results with the comparison points used above (e.g. METIS, FENNEL, etc.), if desired.

For these experiments, we found that ρ (balance) remains at values between 1.00 and 1.01. This is because Leopard considers replication simultaneously with partitioning and therefore maintains the same balance guarantees whether or not replication is used. We therefore only report on how changing the average number of replicas for each vertex affects edge cut ratio. These results are presented in Figure 13. To generate this figure, we enforced a minimum of 2 replicas for each vertex and varied the average number of replicas from 3 to 5. For comparison, the first point in the graph shows the edge-cut without replication (which can be thought of as a minimum and average replica count of 1).

As expected, replication greatly reduces the cut ratio, despite our conservative definition of "edge-cut" in the presence of replication presented above. Even the notoriously difficult to partition Twitter social graph yields an order of magnitude improvement in edge-cut, and with an average of 3 replicas per vertex. However, the marginal benefit of replication drops dramatically as the average number of copies increases. For the Orkut social graph, the cut ratio reduces to 10% with an average of 3 copies. Increasing to 5 copies only brings the cut ratio further down to around 5%.

6.4 Adding Partitions

In Section 5 we described how Leopard repartitions data when a new partition is added by seeding the new partition with existing vertices and examining neighbors of those seeds for reassignment. We proposed three mechanisms for choosing these seeds: (1) choose them randomly from all partitions, (2) choose them randomly from the largest partition, and (3) choose high degree vertices.

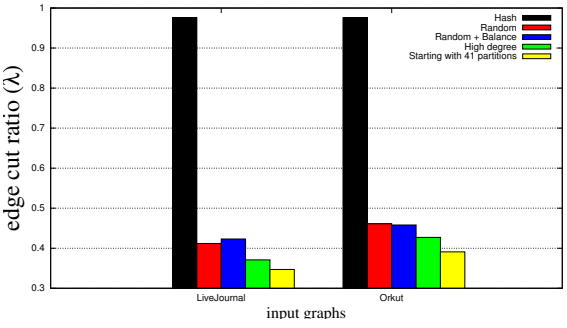

Figure 14: Edge cut after adding a 41st partition

Figure 14 shows how these seeding strategies (which are labeled "Random", "Random + Balance", and "High degree" respectively) affect the quality of partitioning when a new partition is added to the existing 40 partitions from the previous experiments. The number of seeds was set to 5% of the average number of vertices in each partition after

adding a new partition. As a comparison point, we also measure the partitioning if 41 partitions had been used from the beginning. As can be seen, seeding the new partition with high-degree vertices is able to most closely result in a partitioning similar to what the partitioning would have been had 41 partitions been used from the beginning. This is because high degree vertices are often towards the center of a cluster of vertices, and moving clusters intact to a new partition avoids significant disruption of partitioning quality.

7. CONCLUSIONS

In this paper, we proposed a light-weight and customizable partitioning and replication framework for dynamic graphs called Leopard. We studied the effects of reassignment and its timing on the quality of partitioning. By tightly integrating partitioning of dynamic graphs with replication, Leopard is able to efficiently achieve both fault tolerance and access locality. We evaluated our proposed framework and found that even without replication, Leopard consistently produces a comparable cut ratio to statically repartitioning the entire graph after many dynamic updates. However, once replication is integrated with partitioning, the edge cut ratio improves dramatically.

Acknowledgements This work was sponsored by the NSF under grant IIS-1527118.

8. REFERENCES

[1] K. Andreev and H. Räcke. Balanced graph partitioning. In *ACM Symposium on Parallelism in Algorithms and Architectures '04*.

[2] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[3] S. T. Barnard. Pmrsb: Parallel multilevel recursive spectral bisection. In *Supercomputing '95*.

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[5] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput. '08*.

[6] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1), 2010.

[7] Q. Duong, S. Goel, J. Hofman, and S. Vassilvitskii. Sharding social networks. In *Proc. of WSDM*, 2013.

[8] G. Even, J. S. Naor, S. Rao, and B. Schieber. Fast approximate graph partitioning algorithms. In *SODA '97*.

[9] C. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *DAC 1982*.

[10] M. R. Garey and D. S. Johnson. Computers and intractability. 1979.

[11] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *STOC '74*.

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

[13] J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. In *PVLDB '11*.

[14] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD'11*.

[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM'09*.

[16] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of STOC*, 1997.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 1998.

[18] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 1998.

[19] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 1970.

[20] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD '05*.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[22] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD'10*.

[23] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD '12*.

[24] M. E. Newman. Modularity and community structure in networks. *PNAS*, 2006.

[25] J. Nishimura and J. Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *KDD '13*.

[26] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *HPCN Europe '96*.

[27] 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 '10*.

[28] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: Scalable workload-aware data placement for transactional workloads. In *Proc. of EDBT*, 2013.

[29] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 2007.

[30] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *ICDE '12*.

[31] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 1997.

[32] K. Schloegel, G. Karypis, and V. Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 2002.

[33] K. Schloegel, G. Karypis, and V. Kumar. Sourcebook of parallel computing. chapter Graph Partitioning for High-performance Scientific Simulations. 2003.

[34] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD '13*.

[35] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *KDD '12*.

[36] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proc. of VLDB*, 2005.

[37] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM '14*.

[38] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proc. of WSDM*, 2013.

[39] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *ICDCS '14*.

[40] C. Walshaw, M. G. Everett, and M. Cross. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 1997.

[41] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE '14*.

[42] N. Xu, L. Chen, and B. Cui. Loggp: A log-based dynamic graph partitioning method. *PVLDB*, 2014.