IOGP: An Incremental Online Graph Partitioning Algorithm for Distributed Graph Databases

Dong Dai
Texas Tech University
Lubbock, Texas
dong.dai@ttu.edu

Wei Zhang
Texas Tech University
Lubbock, Texas
X-Spirit.zhang@ttu.edu

Yong Chen
Texas Tech University
Lubbock, Texas
yong.chen@ttu.edu

ABSTRACT

Graphs have become increasingly important in many applications and domains such as querying relationships in social networks or managing rich metadata generated in scientific computing. Many of these use cases require high-performance distributed graph databases for serving continuous updates from clients and, at the same time, answering complex queries regarding the current graph. These operations in graph databases, also referred to as online transaction processing (OLTP) operations, have specific design and implementation requirements for graph partitioning algorithms. In this research, we argue it is necessary to consider the connectivity and the vertex degree changes during graph partitioning. Based on this idea, we designed an Incremental Online Graph Partitioning (IOGP) algorithm that responds accordingly to the incremental changes of vertex degree. IOGP helps achieve better locality, generate balanced partitions, and increase the parallelism for accessing high-degree vertices of the graph. Over both real-world and synthetic graphs, IOGP demonstrates as much as 2x better query performance with a less than 10% overhead when compared against state-of-the-art graph partitioning algorithms.

CCS CONCEPTS

•Information systems →Graph-based database models; DBMS engine architectures; Distributed storage;

KEYWORDS

Graph Database; OLTP; Graph Partitioning

1 INTRODUCTION

Graphs have become increasingly important in many applications and domains such as querying relationships in social networks or managing rich metadata generated in scientific computing [2, 8, 21, 38]. These graphs are typically large, hence hard to fit into a single machine. More importantly, even though some graphs may fit into a single server, they are often accessed by multiple clients concurrently, requiring a distributed graph database to avoid performance bottlenecks. For example, our previous work utilized property graphs to uniformly model and manage rich metadata generated in high performance computing (HPC) platforms [6–8]. The rich metadata graph, as the example shown in [8], might not be particularly large (contains millions of vertices and edges), but still needs a distributed graph database to efficiently serve the highly concurrent graph mutations and queries issued from thousands of clients. In fact, a large number of distributed graph database systems have emerged for such task, like DEX [10], Titan [32], and OrientDB [23].

Similar to relational databases, distributed graph databases are designed to serve continuous updates while simultaneously answering arbitrary queries from many clients. They are different from another important set of systems, namely graph processing engines, like Pregel [20], GraphX [37], and X-Stream [27], which focus on performing individual analytic workloads on the whole graphs quickly. In many cases, existing research does not clearly differentiate them because the line between graph databases and graph processing engines is fuzzy. For instance, most graph databases can deliver graph computations through defining complex graph traversal; and many graph computation engines support analytic queries on dynamic graphs. However, regarding the use scenarios they are designed for, there are significant differences. Specifically, graph databases are designed for online transaction processing (OLTP) workloads like `INSERT`, `UPDATE`, `GET`, and `TRAVEL`. These operations are typically issued concurrently from multiple clients and expected to finish immediately. They normally only operate on a small portion of the graph. On the other hand, graph processing engines are designed for online analytic processing (OLAP) workloads, like running PageRank on the whole graph [24] or finding the community structure of social graph [11]. Those workloads are typically issued once in a while with enough changes made in the graph. They often operate on the whole graph and take a long time to finish. These differences lead to completely distinct performance requirements and also affect the design considerations of graph partitioning fundamentally. In this research, we focus on *graph partitioning algorithms for distributed graph databases*.

The first difference is the acceptable cost of time in graph partitioning. Since graph processing engines run analytic workloads on the whole graph which usually take a long time, they can afford to spend more time in partitioning to accelerate later computations. But, this is not the case for graph databases as each transaction is normally short. They have to finish fast and take effect immediately. The graph partitioning algorithms of distributed graph databases have to make per-transaction, online decision rapidly, whereas the ones for graph processing engines do not.

The second difference is the needed knowledge to partition a graph. In most graph processing engines, when the partitioning starts, the majority of the graph is already known. In fact, many

HPDC '17, June 26-30, 2017, Washington, DC, USA

DOI: http://dx.doi.org/10.1145/3078597.3078606

graph partitioning algorithms heavily rely on such knowledge (e.g., vertex degree and its connectivity) to deliver an optimized partitioning. The best-known examples include METIS [16] and Chaco [14]. Although several recent studies (e.g., LDG [22, 30], Fennel [33]) can partition without knowing the whole graph, some local graph information is still necessary. For example, when a vertex is inserted, most of its connected edges should be known at that time. However, in distributed graph databases, vertices and their connected edges are normally inserted independently and concurrently from multiple clients. When the partitioning happens, it is common that neither the global nor the local graph structure is known. The graph partitioning algorithms should be able to work with limited knowledge about graphs, in which case, the existing partitioning algorithms may not be applicable or effective at all.

The third difference is the measurement of partitioning quality. The graph processing engines mainly run analytic tasks on the whole graph, so they are optimized for the best overall throughput. Most existing graph partitioning algorithms are designed for such a goal, which can be formulated as the k-way partitioning problem: 1) minimize "edge cuts" across partitions to reduce the communication cost; 2) maximize "balance" of partitions to avoid potential stragglers. However, these metrics do not necessarily generate good partitions for distributed graph databases. For example, if a graph consists of k equal size disconnected subgraphs, its best k-way partitioning should be just putting each subgraph to one server to achieve the minimized 'cut' and best 'balance'. However, from graph databases' perspective, if any of these subgraphs contains high-degree vertices, graph traversal starting from these vertices will be significantly slower due to the throughput bottleneck of a single machine. The graph partitioning algorithm should consider metrics for individual OLTP operation instead of the overall throughput.

In this paper, we introduce a new graph partitioning solution, namely *Incremental Online Graph Partitioning (IOGP)*, specifically designed for distributed graph databases. It makes per-transaction, online partitioning decisions instantly while serving individual OLTP operation. It adjusts the partitions incrementally in multiple stages based on the increasing knowledge about the graph. It achieves optimized performance for OLTP workloads like graph mutation and graph traversal comparing to the state-of-the-art practices. The contributions of this work are threefold:

- Propose the first (to the best of our knowledge) incremental (multi-stage), online graph partitioning algorithm for distributed graph databases.
- Design and implement the proposed algorithm that incorporates new vertices and edges instantly with limited resources.
- Conduct extensive evaluations of proposed partitioning algorithm on multiple graph data sets from various domains.

Please note that, even though many graph processing systems tend to accommodate large graphs into a single server to avoid network communications introduced by partitioning the graphs (for example, G-store compresses a trillion-edge graph into 2 TB and processes it using one server [18]), the graph size is not the only fundamental reason for partitioning graphs and deploying distributed graph databases. In many cases, the highly concurrent workloads issued from multiple/many clients, demand a distributed graph database to provide quality service to applications, even though the stored graphs are not that large.

The rest of this paper is organized as follows. Section 2 introduces the background for the proposed algorithm. Section 3 analyzes the performance model for graph databases as the basis of IOGP. Section 4 introduces the overview of the three-stage algorithm. In Section 5, more implementation details are introduced. Section 6 reports the evaluation results. Section 7 concludes this study and plans future work.

2 RELATED WORK

It has been well known that graph partitioning problem is NP-hard [13]. In fact, even the simplest two-way partitioning problem is NP-hard [12]. Hence, current widely-used algorithms are heuristic methods. Among them, one important category is called *multi-level* scheme. Examples include METIS [16], Chaco [14], PMRSB [1], and Scotch [25]. They first coarsen the graphs and cut them roughly into small pieces, then refine the partitioning and finally project the pieces back to the finer graphs. These algorithms can be parallelized for improved performance, such as ParMetis [17] and Pt-Scotch [4]. Although algorithms in this scheme are able to handle large graphs efficiently, they are not designed for dynamic graphs, whose vertices and edges are continuously changing. To apply the multi-level scheme to dynamic graphs, re-partitioning the graphs after a batch of changes is typically needed [28]. However, this re-partitioning is heavyweight (can easily take hours in large graphs [31]) and tends to process a batch of changes instead of transactional workloads on graph databases. In contrast, in this paper, we focus on lightweight online partitioning that conducts partitions while changes are streaming into the databases.

In recent years, several lightweight algorithms have been proposed. They partition a graph while performing a single-pass iteration on the data. They normally use some heuristics to decide where to assign current vertex (and all its connected edges), leveraging the local graph structure about vertex. Typical examples include linear deterministic greedy (LDG) [30] and Fennel [33]. However, as we have described in the previous section, in graph database cases, even such local information may not be available while performing partitioning. Another major drawback of such strategy is that each vertex is only assigned once even it might get new edges later. These new changes may deteriorate the previous partitioning. Although several extensions [22, 34] can partition graphs in several passes or iterations, they still suffer in graph database use cases, where vertices and edges are inserted continuously and independently.

Several recent works have introduced online partitioning algorithms for large-scale dynamic graphs, which are relevant to the proposed IOGP algorithm in this study. Vaquero et. al. [35] partitions the dynamic graphs while the processing workloads are running. It updates existing partitions continuously by migrating all vertices in each super-step of a Pregel-like graph batch computation framework. This introduces significant cost and long delay for handling the graph changes, which are acceptable for batch processing, but do not fit for our case. Leopard [15] proposes a partitioning algorithm and a tightly integrated replication algorithm for large-scale, constantly changing graphs. It borrows techniques

from single-pass streaming approaches like Fennel, but improves it with a carefully designed replication strategy. The limitation of Leopard is that it is specifically designed for read-only graph computations to utilize a replication mechanism. Hence, not only graph database workloads do not fit it, but also many graph analysis tasks are not supported, like an analysis of finding single source shortest path. Compared to those algorithms, IOGP is designed to achieve much better performance on OLTP workloads (like accessing to or traveling from a given vertex).

3 MODELING AND ANALYSIS

3.1 Graph Database Model

In this study, we characterize the distributed graph databases with following features: 1) supporting directed graphs; 2) supporting bi-direction traversal, i.e., a vertex can access both its incoming or outgoing edges; and 3) supporting vertices and edges with queryable properties. In fact, these features are common in existing distributed graph databases.

Figure 1 shows a typical architecture of distributed graph databases. In this architecture, each physical server stores a part or a partition of the whole graph in its local *storage engine*. Servers can talk to each other through a high-speed network, and clients are linked with driver libraries to talk to servers. Since each vertex needs to access both its incoming and outgoing edges to enable bi-direction traversal, the storage engine will keep two edge lists as shown in Figure 1. Each server contains an OLTP execution engine to serve requests from clients. The graph partitioning components in both clients and servers cooperate to deliver partitioning. Based on this generic model, we will analyze key factors of OLTP operation performance, which leads to the design and implementation of IOGP described in the next section.

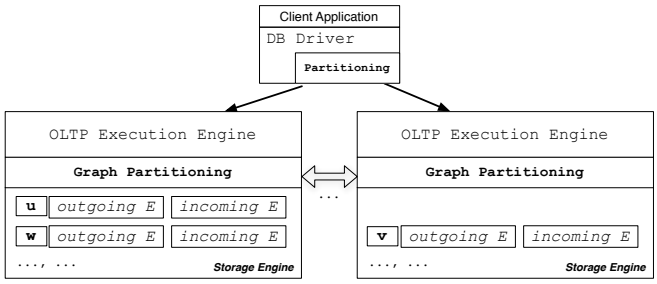

Figure 1: Graph database architecture overview.

3.2 Performance of Single-Point Access

The single-point OLTP operations in graph databases typically include `INSERT`, `UPDATE`, and `GET`. Their performance is largely impacted by whether the clients know the location of the vertex or edge: knowing the accurate location, clients can directly send requests to the server, saving extra cost for querying the location. This could cut the latency by half and double the throughput in many cases. To achieve such a "one-hop" mechanism, clients and servers need to share the same knowledge about current partitions. A widely adopted solution is to use a deterministic hash function, which can be easily shared, to partition the graphs. Many existing distributed graph databases like OrientDB and Titan are using this strategy. Although its drawback is obvious: deterministic hashing does not learn the affinity of vertex connectivity leading to poor locality, its one-hop advantage still deserves considerations for better OLTP single-point access performance. *In this study, the proposed IOGP algorithm maximizes the chance of one-hop access by keeping clients and servers agreeing upon the locations for the majority of the graph.*

3.3 Performance of Graph Traversal

Efficiently supporting graph `TRAVEL` is a unique feature of graph databases and the key difference between graph databases and other storage systems like relational databases or key-value stores [36]. Figure 2(a) shows a sample graph and a traversal starting from vertex u. A traversal usually consists of multiple steps, each contains accesses of vertices and their neighbors in parallel, like those visits of v, w in step S_1.

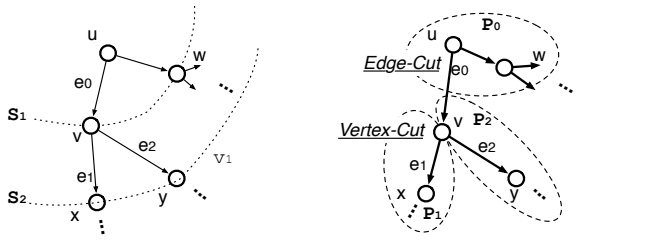

Figure 2: Graph traversal analysis.

Graph partitioning is to place graph vertices and edges into different parts, stored on separate servers. In general, there are two ways to partition a graph as shown in Figure 2(b), i.e., the *edge-cut* and *vertex-cut*. Edge-cut tends to place the source vertex and its connected edges together. Since the destination vertices may be placed on a different server, their in-between edges will look like being cut. For example, u and its neighbors are placed this way and e_0 is cut between two partitions. On the other hand, vertex-cut tends to place the source vertex and its edges separately, so the vertex will look like being cut. For example, v is cut into two partitions as its edges e_1 and e_2 are stored separately shown in the figure.

In fact, regardless of edge-cut or vertex-cut, a 'cut' is introduced as long as two connected vertices are not stored together. For traversal, such a 'cut' simply means extra network communications between servers. Hence, all graph partitioning algorithms strike for minimizing these cuts to achieve better locality between vertices. *In this study, the proposed IOGP algorithm enhances the locality between vertices by leveraging a heuristic method to dynamically adjust vertex location.*

In addition, even with the same locality, vertex-cut and edge-cut can lead to different performance. For instance, if a vertex u has more than one million connected edges, which is highly possible in real-world power-law graphs, edge-cut will store all edges together with u. This will lead to long time for loading

edges while accessing u. Comparatively, vertex-cut can assign these edges into multiple servers to amortize the workloads and deliver much better performance. On the other hand, if a vertex u has a small number of edges, splitting them into multiple servers introduces extra network communications, diminishing the benefit of parallelism. In such cases, which are quite often as most vertices in power-law graphs have a small number of edges, edge-cut is clearly a better choice. *In this study, the proposed IOGP algorithm considers the degree of a vertex during partitioning and chooses the better way to partition graphs accordingly.*

4 ALGORITHM OVERVIEW

The goal of the IOGP is to optimize the performance of OLTP operations in graph databases. The performance analysis in previous sections enlightens and rationalizes its design and implementation. Specifically, IOGP first leverages deterministic hashing to quickly place new vertices. This strategy enables one-hop access for most of the graph vertices by default. While more edges of a vertex are inserted, IOGP will adjust the location of the vertex to achieve better locality leveraging the increasing knowledge about the vertex connectivity. Until this step, the graph is still partitioned following the *edge-cut* partitioning. However, once a vertex has too many edges, IOGP will apply *vertex-cut* to increase the parallelism and further improve the traversal performance. In this way, IOGP manages to generate high-quality partitions while serving continuous OLTP operations. We summarize IOGP into three stages, namely *quiet stage*, *vertex reassigning stage*, and *edge splitting stage* respectively, and introduce them in more details below.

4.1 Quiet Stage

IOGP operates in quiet stage by default. At this stage, it places a new vertex into a server using the deterministic hashing function. All clients and servers share the same function to ensures the one-hop access. Following *edge-cut*, IOGP places new edges together with their incident vertices. Note that an edge $u \rightarrow v$ will be stored in both the outgoing edge list of u and the incoming edge list of v to enable the bi-direction graph traversal.

The problem of deterministic hashing is it does not consider the locality affinity of vertices. It is not a significant problem when a vertex does not have many edges, but would lead to problems while vertex grows. IOGP solves the problem in the *vertex reassigning stage* after knowing more about the vertex connectivity. In addition, as edge-cut may create hotspots if the vertices have too many edges, IOGP applies vertex-cut in the *edge splitting stage* to address this issue.

4.2 Vertex Reassigning Stage

In the quiet stage, vertices do not have enough connectivity information, hence random hashing is a good option. But, as more edges are inserted, more connectivity information is obtained. It is desired to leverage such knowledge to re-assign vertices to a better partition. The goal is straightforward: move a vertex to a partition that stores most of its neighbors while keeping all partitions balanced to avoid stragglers.

To determine which partition is the best choice, IOGP leverages the Fennel heuristic score [33], as shown in Equation 1. Here, 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 adjustable parameters.

$$max\{|N(v) \cap P_i| - \alpha \frac{\gamma}{2}(|P_i|)^{\gamma-1}\} \quad (1)$$

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

In Fennel, such a heuristic score is calculated simply by scanning all neighbors of the vertices in each partition. The time cost is acceptable as Fennel is not designed for serving OLTP operations. However, such computation consumes too much time in our focused cases. To solve this issue, in this research we propose a new strategy to calculate it by maintaining edge counters continuously. More details are introduced in Section 5.

4.3 Edge Splitting Stage

In a power-law graph, degree of a vertex could be extremely large. As we have discussed, the *edge-cut* may lead to significant performance degradation. In IOGP, we introduce the edge splitting stage to handle it. Specifically, we propose to split edges of high degree vertices into multiple servers to amortize the loads. In the generic graph database model, each vertex contains incoming edges and outgoing edges. We consider them together as traversals may happen in both directions.

IOGP defines a threshold `MAX_EDGES` to decide when to split a vertex. If a vertex degree exceeds this number, IOGP will cut and split all its edges. The splitting is quite simple: IOGP will place an outgoing edge together with its destination vertex and place an incoming edge together with its source vertex. Figure 3 shows an example of splitting with three storage servers. In this example, u's edges need to be split to offload its loads. Initially, all edges (from 1 to 6) are stored with u on server 1. After splitting, they are assigned across all three servers according to the locations of their destination vertices. Note that the vertex u is not moved. The ones on server 2 and 3 are just Id index (shown in shadow pattern in the figure).

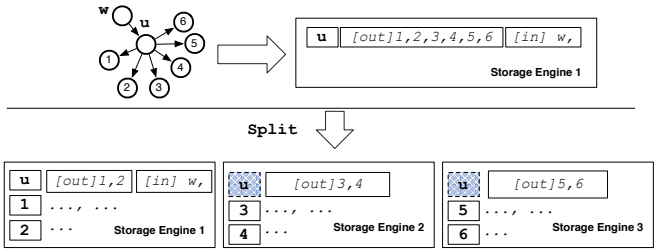

Figure 3: An edge splitting example

The locality does not change in this stage because an edge is moved to either its source or destination vertex without altering the locality. However, this will significantly improve the performance of accessing a high-degree vertex as these operations can be carried out in parallel across multiple servers. Also, concurrent edge mutations on that vertex can be offloaded to multiple servers for better performance.

5 ALGORITHM DESIGN AND IMPLEMENTATION

In the previous section, we briefly describe the three stages of IOGP. However, its implementation in distributed graph databases is non-trivial. A number of implementation challenges and various design trade-offs remain. In this section, we will introduce more design and implementation details.

5.1 IOGP Data Structure

IOGP introduces a series of data structures to achieve efficient online graph partitioning. These data structures are mainly counters, which record the states of vertices in each partition. They are stored in memory for quick access. In case of failures, they can be rebuilt from a full scan on the existing database.

- On the server currently storing vertex v, there is a $split(v)$ indicating whether its edges have been split or not.
- On the two servers that originally or currently store vertex v respectively, a $loc(v)$ records its accurate location. It only exists once IOGP reassigns the vertex, serving as a location service for the graph database.
- Each vertex v has maximum four edge counters to incrementally maintain its connectivity information. These counters may be stored on multiple servers.
  1) $alo(v)/ali(v)$ store the number of *actual local outgoing/incoming* edges of v. They count the outgoing/incoming edges whose destination/source vertices are also stored in local server, i.e., local neighbors. They only exist in server that actually stores v.
  2) $plo(v)/pli(v)$ store the number of *potential local outgoing/incoming* edges of v. These two counters exist in servers that do not store v. They count v's local neighbors if v has been moved back to local server.
- Each server also maintains a *size* counter, indicating its vertices and edges number.

Overall, those data structures are small. Each server only has one *size* counter. For each vertex v, the $split(v)$ and $loc(v)$ only exists on one or two servers, hence also scales well. But, the edge counters may exist on all servers: one server stores alo, ali and all others store plo, pli. If each counter takes 2 bytes, together they take 4 bytes per vertex on each server. This might lead to a problem if the entire graph database stores over a billion vertices, which will consume over 4GB memory on each server in the worst case. However, the real cases are much better than this worst scenario for two reasons: 1) vertices that enter *edge splitting stage* do not need edge counters anymore, and 2) the $plo(v), pli(v)$ potential counters only exist in servers that store v's neighbors. These significantly reduce the memory consumption in real-world power-law graphs. In the evaluation section, we show more details about the memory footprints of these counters.

5.2 Quiet Stage Implementation

In the quiet stage, IOGP places vertices using the deterministic hashing function by default. Note that to support bi-direction traversal, inserting an edge like $e(u \rightarrow v)$ will lead to two insertions: one as the outgoing edge of u and the other as the incoming edge of v.

IOGP maintains *edge counters* for vertex reassignment. Initially, we set all counters to 0. Once a new edge $(u \rightarrow v)$ is inserted, two insertions are issued. On the server that stores the source vertex (s_u), after successfully inserting the edge as the outgoing edge of u, IOGP will check whether the destination vertex v is also stored locally. This check can be done instantly by examining the hash value of v and the existence of $loc(v)$ in local memory. If yes, the edge is local to both its source and destination vertices, hence it increases $alo(u)$ by 1 as this indicates the existence of actual locality. If not, it increases $pli(v)$, which means only potential locality is introduced for v. Note that, this pli counts for vertex v, which means that only v is moved back to this server in the future, then the actual locality can be obtained. Similarly, on the server that stores the destination vertex (s_v), counters are updated accordingly.

IOGP updates edge counters while serving vertex and edge insertions. The actual local edges (alo, ali) and potential local edges (plo, pli) are used in the vertex reassigning stage to calculate the best partition for a vertex efficiently.

5.3 Vertex Reassigning Stage Implementation

In the vertex reassigning stage, IOGP tries to reassign the vertex to a different server to enhance the locality. The first task of reassigning vertex is to calculate the best partition. According to the description in Section 4.2, instead of scanning the databases to obtain $|N(v) \cap P_i|$, IOGP leverages the edge counters to efficiently calculate the best location.

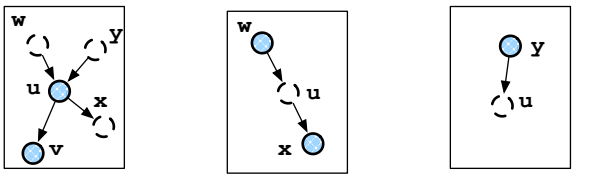

Figure 4: An example of partitions during vertex reassignment. Edge counters are shown.

Figure 4 shows a sample graph with 5 vertices and edges, partitioned into three servers. We also show their edge counters. Here, solid circles with colored patterns indicate actual existence of vertices in that server; dashed circles indicate the vertices do not exist,

only their edges exist. As this figure shows, each edge actually is stored twice. For example, $e(u \rightarrow x)$ is stored in $server_1$ as an outgoing edge of u, and at the same time, stored in $server_2$ as an incoming edge of x.

In this example, only edge $e(u \rightarrow v)$ indicates the actual locality, which means that we have $alo(u) = 1$ and $ali(v) = 1$ on $server_1$. The other three edges only indicate the potential locality. The relevant edge counters are shown in Figure 4. These values are efficiently maintained in the quiet stage.

When IOGP reassigns a vertex, like u, it will compare whether moving u to another server will increase or decrease the score calculated from Equation 1. Specifically, moving u out of s_1 will certainly reduce the amount of locality on server s_1 by $2*(alo(u)_{s_1} + ali(u)_{s_1})$. We double it because the locality decrements come from both vertex u and its locally connected vertices. At the same time, moving u into another server s_j will increase its locality by $2 * (plo(u)_{s_j} + pli(u)_{s_j})$. The partition size *size* on each server should also be calculated. IOGP will choose the partition s_i that obtains the largest positive value from following equation:

$$\begin{aligned} ra\_score_{s_i} = max\{&2 * (plo(u)_{s_i} + pli(u)_{s_i}) \\ &- 2 * (alo(u)_{s_{cur}} + ali(u)_{s_{cur}}) \\ &+ [size_{s_i} - size_{s_{cur}}]\} \end{aligned} \tag{2}$$

This equation is derived from Equation 1 by choosing parameter $\alpha = 1$ and $\gamma = 2$. These parameters are also widely used in existing studies [15]. If we take Figure 4 as an example, vertex u would be reassigned to $server_2$ as its *ra_score* is 1.

5.3.1 Maintain IOGP Data Structure. Algorithm 1 shows how IOGP maintains the in-memory data structures while reassigning a vertex. When a vertex u is moved, the $loc(u)$ in the original server will be updated to its new location. Any further reassignment also updates the $loc(u)$ in the original server. This serves as a distributed location service for the graph database. A fresh client needs to ask the original server that stores u to retrieve its current location through querying $loc(u)$. Clients can cache the location for future requests. In addition, servers involved in this reassignment will update their *size* counter accordingly.

In terms of updating the edge counters, vertex u's counters are updated first: 1) in the original server s_u, u's actual locality will turn into a potential locality; 2) on the target server s_k, u's potential locality will turn into an actual locality. In addition to updating u, it is more important to update vertices that are connected to u. Their actual localities are changed because vertex u is moved out or in. For example, in the original server (s_u), for all u's incoming edges, if their source vertices (*src*) are also stored in local server, we need to reduce their actual outgoing locality ($alo(src)$) by 1 because their destination vertex u is no longer in local server. This is also required for outgoing edges. The target server s_k performs similar updates except it will increase the localities. More importantly, every time a vertex u is reassigned, the edge counters of its neighbors also need to be updated. These updates are actually fast (as iterating u's incoming and outgoing edges in-memory) and overlapped with the actual data movement (described in Section 5.5).

5.3.2 Timing of Vertex Reassignment. The timing of reassigning vertices is critical to balance partitioning quality and overheads.

Algorithm 1 Maintain IOGP Data Structure

```
1:  ◇ Assign u from s_u to s_k
2:  if on server s_u then                          ▷ on source server s_u
3:      size -= 1;
4:      plo(u) = alo(u);
5:      pli(u) = ali(u);
6:      for e ∈ incoming(u) do
7:          if e.src stored in s_u then
8:              alo(e.src) -= 1;
9:      for e ∈ outgoing(u) do
10:         if e.dst stored in s_u then
11:             ali(e.dst) -= 1;
12:
13: if on server s_k then                          ▷ on target server s_k
14:     size += 1;
15:     alo(u) = plo(u);
16:     ali(u) = pli(u);
17:     for e ∈ incoming(u) do
18:         if e.src stored in s_k then
19:             alo(e.src) += 1;
20:     for e ∈ outgoing(u) do
21:         if e.dst stored in s_u then
22:             ali(e.dst) += 1;
```

This is especially true for the proposed online IOGP algorithm. We have observed that when a vertex has more edges, its connectivity becomes more stable, thus less reassignment is needed. This rationale is rather straightforward. For example, when a vertex has only one edge, a new edge may significantly change its locality affinity. But, if a vertex has 1K edges already, most likely a new edge does not make a significant difference. This observation and rationale lead to our design in IOGP: 1) deferring vertex reassignment until its connectivity stabilizes; and 2) reducing vertex reassignment frequency while more edges are inserted. Specifically, we consider until a vertex contains over `REASSIGN_THRESH` connected edges (both incoming and outgoing edges), a vertex reassignment attempt can be made. After a reassignment, we will check the possibility of another reassignment only after a similar amount of new edges are inserted. Assuming k=`REASSIGN_THRSH`, we check vertex reassignments when it reaches $[k, 2 * k, 4 * k, .., 2^i * k, ..]$ edges. This significantly reduces the number of reassignments for a vertex. For example, if `REASSIGN_THRSH`=10, for a vertex with 10,240 edges, the maximum number of movements is only 10. The choice and impact of `REASSIGN_THRSH` will be discussed in the evaluation section.

5.4 Edge Splitting Stage Implementation

The edge splitting stage is a key optimization of IOGP for high-degree vertices. It is mainly designed to amortize loads of accessing high-degree vertices and to improve the performance of operations like scan and traversal.

As described in the vertex reassigning stage, when a vertex is split, it may have already been reassigned multiple times. But, once a vertex enters into the splitting stage, it will never be reassigned again. IOGP will invalidate and free up all its edge counters to reduce the memory footprint. This strategy is chosen for two reasons. First, when a vertex is split across the cluster, statistically, its edges will be evenly distributed as their neighbors are randomly

distributed through hashing. Hence, reassigning vertex will not significantly increase the locality anymore. Second, moving vertices that have been split also introduces unnecessary complexity. The algorithm needs to take extra care when a vertex is reassigned and its edges are already split, which may invalidate the edge counters.

Regarding updating the IOGP data structures, it is straightforward in the edge splitting stage. First, it updates $split(u)$ to the corresponding value. Second, it invalidates and frees up local edge counters of vertex u. It further frees up edges counters of u in other storage servers along with the edges movement. The sizes of u's incoming and outgoing edges will be updated accordingly.

5.5 Asynchronous Data Movement

In an IOGP-enabled graph database, there are two extra data movements introduced: vertex reassigning and edge splitting. Moving data synchronously while serving OLTP requests can cause potential performance issues. In IOGP, we optimize these data movements to be asynchronous to avoid blocking OLTP operations.

During edge splitting, once IOGP needs to split a vertex, it will update the in-memory IOGP data structures and add the vertex into the *pending splitting queue* in one transaction. Once this transaction finishes successfully, even without moving data yet, we start to reject new edges that should not be stored locally. Clients that issue edges insertions to a wrong server will be rejected with a notification indicating that the vertex has been split. Clients synchronize their statuses based on the replies and request the correct server again. Reassigning vertices is similarly handled. After determining the target server, it will update in-memory IOGP data structures, and then add the vertex into the *pending reassigning queue* in one transaction. The server will also stop serving requests about the vertex and notify clients to request the target server in the future. For both cases, the real data movement actions are implemented via a background thread, which periodically retrieves vertex v from the header of pending queues and handles the data movement for it. After data has been moved, the local copy will be removed afterward.

This asynchronous data movement mechanism is efficient, but may introduce a problem for read requests because the requested vertices or edges may be in an uncertain status while data movement takes place. They could be on the original server (copying is not started yet), on the new server (copying and deleting are finished already), or even on both of them (copying is finished but not deleting). To solve this, the clients need to issue *two* read requests concurrently for elements that are under movement: one request is sent to the original server, and the other one is sent to the new server. If both requests get results, the one from new server wins. Clients can learn whether the edge movement has finished or not based on the replies from new servers and avoid the extra requests in the future.

6 EVALUATION

6.1 Evaluation Setup

All evaluations were conducted on the CloudLab APT cluster [5]. It has 128 servers in total, and we used 32 servers as the back-end servers. Each server has an 8-core Xeon E5-2450 processor, 16GB RAM, and 2 TB local hard disk. All servers are connected through 10GbE dual-port embedded NICs. Unless explicitly stated, we used all 32 servers in experiments.

6.1.1 Dataset Selection. We used the popular SNAP dataset for real-world graph evaluations [19]. SNAP is a collection of networks from various domains, and most of them are power-law graphs. We show a representative selection of these graphs used in our evaluations and outline their properties and scales in Table 1.

Specifically, we selected graphs scaling from less than 200K edges to almost 100M edges to represent different stages of continuously growing graphs that graph databases serve. Although many graph processing frameworks are capable of processing graphs with these sizes (i.e., the number of edges or vertices) in a single server, we do consider distributed graph databases are still necessary for these graphs in practice. As our previous work has shown [6–8], a graph with millions of vertices and edges may be accessed by thousands of clients concurrently, hence demands graph partitioning and a distributed graph database solution. Additionally, the property graphs tend to have a rich set of queryable properties. They can easily be large enough (e.g., multiple KB) to make a graph with millions of vertices and edges not fit for a single machine.

In this evaluation, another reason we did not include tremendously large graphs is, unlike the offline graph partitioning algorithms or the underlying storage engines, the online algorithms like IOGP, are not sensitive to the size of the graph. Instead, they concentrate on the structures of the graphs (e.g., the connectivity). So we considered a diverse set of structures when selecting graphs from various domains in the datasets. Note that the SNAP dataset only contains graph structures. We attached randomly generated property, a 128K bytes key-value pair, on each vertex and edge.

Table 1: Selected graphs from SNAP dataset

Data Set	*Domain*	*Vertex Num.*	*Edge Num.*
as-Skitter	network	1,696,415	11,095,298
web-Google	web	875,713	5,105,039
roadNet-CA	geo	1,965,206	2,766,607
Loc-Gowalla	geo	196,591	950,327
amazon0302	purchase	262,111	1,234,877
amazon0601	purchase	403,394	3,387,388
ca-AstroPh	social	18,772	198,110
wiki-talk	social	2,394,385	5,021,410
email-EuAll	social	265,214	420,045
email-Enron	social	36,692	183,831
soc-Slashdot0902	social	82,168	948,464
Soc-LiveJournal1	social	4,847,571	68,993,773
cit-Patents	citation	3,774,768	16,518,948
cit-HepPh	citation	12,008	118,521

We also used synthetic graphs to evaluate IOGP. The synthetic graphs were generated using the RMAT graph generator [3] following the power-law distribution. We used the following parameters to generate an RMAT graph with 10K vertices and 1.2M edges: $a = 0.45, b = 0.15, c = 0.15, d = 0.25$. The graph is named as *RMAT-10K-1.2M*.

6.1.2 Software Platform. We evaluated IOGP on a distributed graph database prototype, namely SimpleGDB [29]. Its core has

been used in several research projects and proven to be efficient [6, 7]. More importantly, its flexible design supports various graph partitioning algorithms and enables fair comparison among them.

SimpleGDB follows the generic graph database architecture shown in Figure 1. It uses consistent hashing to manage multiple storage servers in a decentralized way by mirroring Dynamo's approach [9]. This allows the dynamic growth (or shrinking) of the graph database cluster. Each server runs the same set of components including an OLTP execution engine, a data storage engine, and a graph partitioning layer. The OLTP execution engine accepts requests from clients and serves them. The storage engine organizes graph data such as vertices, edges, and their properties into key-value pairs and stores them persistently in RocksDB [26]. The graph partitioning layer is designed as a plugin to allow hackers to change algorithms without affecting other components, which largely simplifies the evaluation and the fair comparisons presented in this study. Another key feature of SimpleGDB is that it contains a server-side asynchronous graph traversal engine built based on study [6]. Through a server-side traversal, we are able to fully utilize the locality gained by graph partitioning algorithms.

6.2 Evaluation Results

6.2.1 Edge-Cut and Balance. We first compare the k-way partition metrics (i.e., edge cuts and partition balance) among IOGP and the state-of-the-art graph partitioning algorithms (METIS, Fennel, and Hash). Since METIS cannot efficiently work with OLTP workloads, to conduct the comparison, we actually ran METIS on the final graph once, assuming all vertices and edges were already inserted. Similarly, to conduct the fair comparison against Fennel, we assume that the graph is inserted in a way that a vertex and all its edges are inserted together. Their insertion order is chosen randomly. Results of the hashing and IOGP were conducted in an online manner following the same order as the datasets provided.

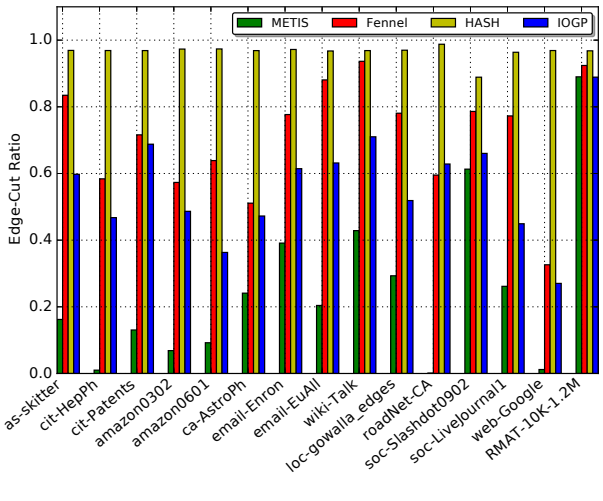

Figure 5: Edge-cut ratio comparison.

We plot the results of all graphs (described in the previous subsection) in Figure 5 and 6. Figure 5 shows the edge-cut ratio, calculated as the number of edge cuts over the total number of edges in a graph. Figure 6 shows the imbalance ratio, calculated as the maximum difference among all partitions over the average partition size. Since Fennel, IOGP, and Hash achieve highly balanced partition, their imbalance ratios are almost zero for all cases. Their results cannot be seen in the figure. From these results, we have several observations. First, METIS achieves the best locality but worst balance among all tested algorithms. In the *web-Google* graph, it results in a partition with less than 1% edge-cut ratio, but over 6% imbalance. On the other hand, Hash results in the worst partitioning in all cases, but at the same time, provides excellent balance. Second, IOGP and Fennel are in between of METIS and Hash and their imbalance is small. In terms of edge-cut ratio, IOGP is better than Fennel in all tested cases. In many cases (e.g., *email-EuAll* and *wiki-Talk*), the difference is clear. These results confirmed that IOGP can obtain better vertex locality than the state-of-the-art streaming partitioning algorithms like Fennel, even using the same heuristic functions. The reason is quite straightforward. Fennel only assigns a vertex once when it is first inserted. But, IOGP may reassign a vertex multiple times during continuous insertions and hence have more chances to choose a better location for a vertex. We will show more detailed analysis in the next subsection.

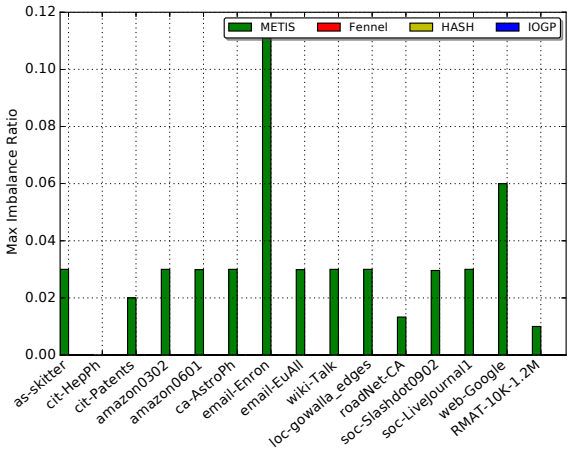

Figure 6: Partitions balance comparison.

6.2.2 Continuous Refinement of IOGP. As shown from the evaluations reported and discussed in the previous sub-section, IOGP achieves better locality than Fennel due to its ability to continuously refine the partitions. In Figure 7, we show how this happens in detail. The x-axis indicates the number of insertions that happened during constructing the graph. The y-axis shows current edge-cut ratio. We took a sample after every 10^5 insertions. We show the first $2 * 10^7$ insertions in this figure. The results confirm two important patterns that we leverage in IOGP: 1) the initial insertions changed the locality more significantly, and 2) graph becomes more stable while more edges are inserted. This is also why IOGP is designed to increase the `REASSIGN_THRSH` exponentially to reduce the frequency of reassignment.

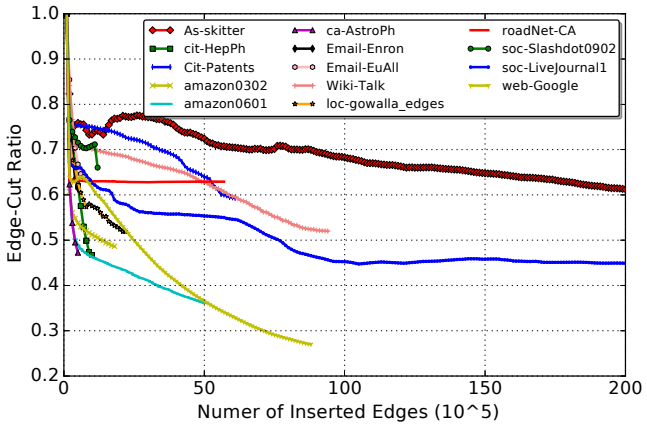

Figure 7: Changes of edge-cut ratio while inserting.

6.2.3 Vertex Reassigning Threshold. We discuss the reassignment threshold (i.e., REASSIGN_THRSH) in this evaluation. Specifically, we constructed the whole graph multiple times using different reassignment thresholds and collected the edge-cut ratio of each round and the number of vertex reassignments. It is expected that a smaller REASSIGN_THRSH brings more overheads (i.e., more vertex reassignments), and generates better partitions (i.e., smaller edge-cut ratio). In fact, the best value for REASSIGN_THRSH should be different for separate graphs. In this evaluation, we tested a wide range of possible values to find the potential rules in choosing this value. Specifically, we iterated thresholds from 1 to 50 with an increase of 5 each step. All results are plotted in Figure 8.

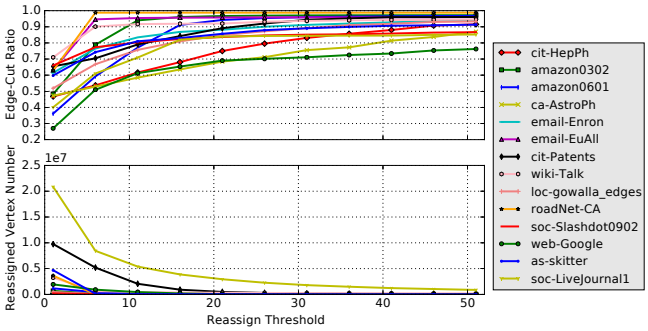

Figure 8: Edge-cut ratio and reassignment times.

The top sub-figure shows that the edge-cut ratio increases as the REASSIGN_THRSH become larger. More specifically, the increase is significant at the beginning and turns into flat afterward. This is because most of these graphs have a small average degree (according to Table 1), and they are more sensitive to threshold changes in the smaller end. Once the threshold became sufficiently large, their ratios became more stable. In the bottom sub-figure, we show how many times of the vertices are reassigned with different thresholds. As expected, a larger threshold reduces the number of vertex reassignments. From those results, we conclude that the best choice of REASSIGN_THRSH should be near half of the average degree of the graph to strike a balance between achieving better locality and less vertex reassignments. This is an empirical result, like, 6 for *web-Google*.

6.2.4 Edge Splitting Threshold. In IOGP, we split a vertex based on its degree to achieve the best traversal performance in the edge splitting stage. Although splitting edges into multiple servers saves time while loading data from disks, it does introduce extra network overhead to retrieve data from remote servers. It is important to find the best threshold to balance the disk and network latency. As we have described, the splitting threshold is relevant with both the hardware (disk speed and network latency), the scale of the distributed cluster, and the vertex degree. It is non-trivial to obtain a universally optimal setting. In this evaluation, we aim to build a general guideline of choosing the edge splitting threshold. It is desired to conduct similar evaluations before deploying IOGP on a specific system to obtain the optimal setting.

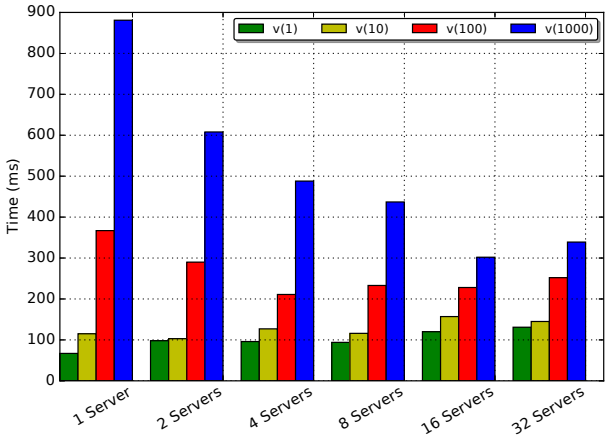

Figure 9: Scan performance with different degrees.

Specifically, we conducted a series of evaluations on various cluster scales (from 2 → 32 servers), towards different vertices with distinct degrees (from 1 → 10^3). Each edge is attached with 128KB randomly generated properties. The disk and network latency are fixed based on the hardware configuration of CloudLab APT cluster. For comparison, we measured the time cost of one-step traversal from these vertices in different cluster scales. The results are reported in Figure 9. The x-axis shows different scales in the evaluations, where 'k server(s)', indicates all edges are split among all of them. Note that the case of '1 server' means there is no edge-splitting. The y-axis shows the time cost of reading each vertex and its neighbors. There are four cases in total. From these results, we can draw several observations. First, low-degree vertices like $v(1)$ and $v(10)$ tend to obtain better traversal performance in smaller scale cluster. On the other hand, high-degree vertices achieve better performance in larger scale cluster. This also confirms our previous analysis. Second, each degree has its best scale. For example, for a vertex with 10^3 edges, the minimum time is obtained in '16 servers' cluster. For a vertex with 100 edges, '4 servers' cluster would be

the best. This metrics can guide the deployment to choose the best MAX_EDGES for a specific cluster.

6.2.5 Memory Footprint of IOGP Data Structure. As we have discussed in Section 5, IOGP introduces a number of in-memory counters to facilitate partitioning process. Their memory footprints may limit the scalability of IOGP algorithms. In this evaluation, we examined the maximal memory footprint during constructing the graphs listed in Table 1. The results are plotted in Figure 10. The x-axis shows different graphs and the y-axis shows the maximal memory consumption (KB) across 32 servers. We also plot the 'Expected' memory footprint, which is calculated simply assuming each vertex v has two edge counters in each server. From these results, we can easily observe that, the actual memory consumption is much smaller than the upper-bound estimation, especially for those large-scale graphs. These results from real-world graphs clearly show that IOGP is practical in partitioning large-scale graphs.

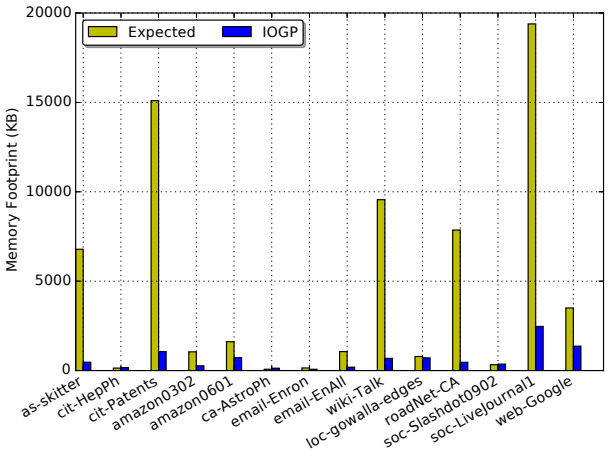

Figure 10: Memory footprint of IOGP.

6.2.6 Single-point Access Performance. As we have described, most graph databases use simple hashing strategy to deliver online graph partitioning. Hashing is fast and benefits single-point OLTP operations like INSERT most. Other graph partitioning algorithms including METIS and Fennel are expected to have much worse performance on insertions due to their offline nature. In this research, to study the benefit of IOGP, we compared its insertion performance with the best algorithm (hashing). Again, the evaluations were conducted in the 32-server SimpleGDB cluster. Figure 11 plots the insertion speed of IOGP and Hash algorithms. The performance was generated from a single client. As the results show, Hash always performs better than IOGP as expected, because there are overheads introduced by vertex reassigning and edges splitting. However, the difference is small and less than 10%.

6.2.7 Graph Traversal Performance. In this evaluation, we further compared the traversal performance of IOGP and Hash. As the most important OLTP operation in graph databases, graph traversal should obtain the best performance. This is achieved by less edge-cut ratio between reassigned vertices and higher parallelism

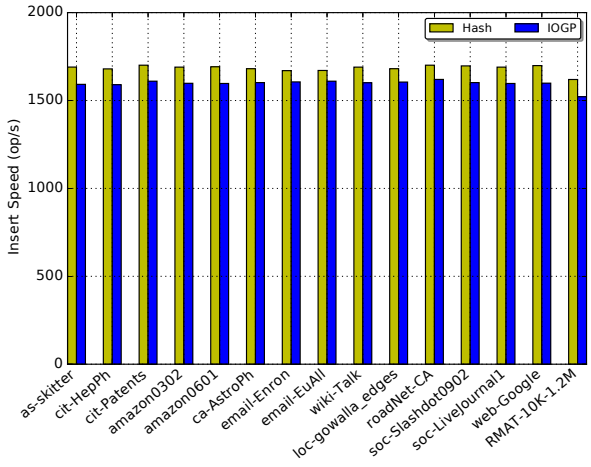

Figure 11: Insertion performance.

while accessing split high degree vertices. In this evaluation, all traversals started from the same set of randomly chosen vertices. Their average finish time is used for comparison. We evaluated graph traversal with 2, 4, 6, and 8 steps.

Due to the space limit, we cannot show the comparison results from all tested graphs. Instead, we chose a set of representative graphs based on the edge-cut ratio shown in Figure 5. Specifically, we selected two graphs that have the maximal edge-cut ratio difference between Fennel and IOGP (i.e. *web-Google* and *RMAT-10K-1.2M*) and two graphs that have the minimal edge-cut ratio difference (i.e. *soc-LiveJournal1* and *wiki-Talk*). We excluded METIS since it is not valid in streaming graphs to avoid unfair comparison.

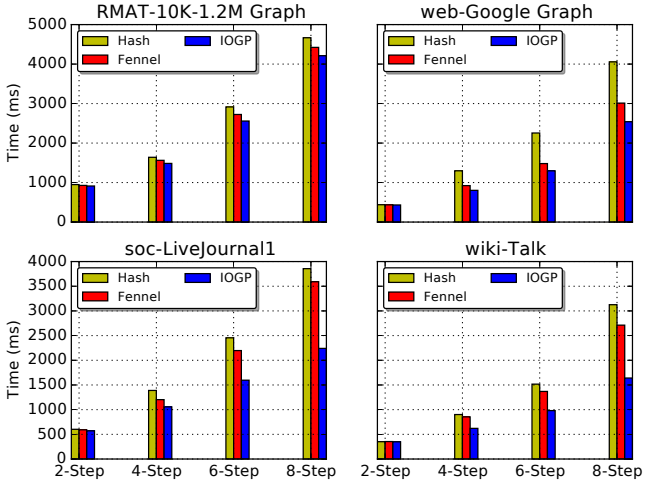

Figure 12: Graph traversal performance.

The results are plotted in Figure 12. As the results show, IOGP achieves clearly better traversal performance than Hash and Fennel for all cases. The performance gap also increases while more traversal steps are performed. These results demonstrate the advantage

and importance of IOGP for future, more complex graph traversal requests. Additionally, we can observe that IOGP achieves more improvements on graphs with better edge-cut ratio. This observation recalls the importance of vertex locality in graph partitioning.

7 CONCLUSION & FUTURE WORK

In this study, motivated by the OLTP performance requirements of distributed graph databases, we have introduced an Incremental Online Graph Partitioning (IOGP) algorithm and have described its design and implementation details. IOGP adapts its operations among three stages according to the continuous changes of the graph. It operates fast, obtains optimized partition results, and generates partitioned graphs serving complex traversals well. We have also presented implementation details including in-memory data structures (e.g., edge counters) to deliver fast, online graph partitioning. Our detailed and concrete evaluations on multiple graphs from various domains confirmed the advantages of IOGP. From these evaluations, we are also able to draw important conclusions including the general guidelines of selecting its key parameters. We believe that IOGP has the great potential to be widely used as a graph partitioning solution for distributed graph databases. In the future, we plan to investigate and develop fault tolerance feature for IOGP, with a focus on rebuilding in-memory data structures efficiently when needed.

8 ACKNOWLEDGMENTS

We are thankful to the anonymous reviewers for their valuable feedback and our shepherd, Dr. Jay Lofstead, for his detailed and valuable suggestions that improved this paper significantly. This research is supported in part by the National Science Foundation under grant CNS-1162488, CNS-1338078, IIP-1362134, and CCF-1409946.

REFERENCES

[1] Stephen T Barnard. PMRSB: Parallel Multilevel Recursive Spectral Bisection. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*.
[2] Peter J Carrington, John Scott, and Stanley Wasserman. 2005. *Models and methods in social network analysis*. Vol. 28. Cambridge university press.
[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, Vol. 4. SIAM, 442–446.
[4] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6 (2008), 318–331.
[5] CloudLab. 2017. https://www.cloudlab.us/. (2017).
[6] Dong Dai, Philip Carns, Robert B Ross, John Jenkins, Kyle Blauer, and Yong Chen. 2015. GraphTrek: Asynchronous Graph Traversal for Property Graph-Based Metadata Management. In *2015 IEEE International Conference on Cluster Computing*. IEEE, 284–293.
[7] Dong Dai, Yong Chen, Philip Carns, John Jenkins, Wei Zhang, and Robert Ross. 2016. GraphMeta: A Graph-Based Engine for Managing Large-Scale HPC Rich Metadata. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*. IEEE, 298–307.
[8] Dong Dai, Robert B Ross, Philip Carns, Dries Kimpe, and Yong Chen. 2014. Using property graphs for rich metadata management in hpc systems. In *Parallel Data Storage Workshop (PDSW), 2014 9th*. IEEE, 7–12.
[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. (2007).
[10] DEX. 2017. DEX. http://www.sparsity-technologies.com/. (2017).
[11] David Ediger, Jason Riedy, David A Bader, and Henning Meyerhenke. 2011. Tracking structure of streaming social networks. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, 1691–1699.
[12] Michael R Garey and David S Johnson. 2002. *Computers and intractability*. Vol. 29. wh freeman New York.
[13] Michael R Garey, David S Johnson, and Larry Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*. ACM, 47–63.
[14] Bruce Hendrickson and Robert Leland. 1995. *The Chaco user's guide: Version 2.0*. Technical Report. Technical Report SAND95-2344, Sandia National Laboratories.
[15] Jiewen Huang and Daniel J Abadi. 2016. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment* 9, 7 (2016), 540–551.
[16] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.
[17] George Karypis and Vipin Kumar. 1998. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel and Distrib. Comput.* 48, 1 (1998), 71–95.
[18] Pradeep Kumar and H Howie Huang. 2016. G-store: high-performance graph store for trillion-edge processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 71.
[19] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).
[20] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.
[21] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray User's Group (CUG)* (2010).
[22] Joel Nishimura and Johan Ugander. 2013. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD*. ACM, 1106–1114.
[23] OrientDB. 2017. http://www.orientechnologies.com/orient-db.htm. (2017).
[24] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).
[25] François Pellegrini and Jean Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking*. Springer.
[26] RocksDB. 2017. http://rocksdb.org/. (2017).
[27] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-Centric Graph Processing using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.
[28] Kirk Schloegel, George Karypis, and Vipin Kumar. 1997. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel and Distrib. Comput.* 47, 2 (1997), 109–124.
[29] SimpleGdb. 2017. https://github.com/daidong/simplegdb-Java. (2017).
[30] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*.
[31] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.
[32] Titan. 2017. http://thinkaurelius.github.io/titan/. (2017).
[33] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 333–342.
[34] Johan Ugander and Lars Backstrom. 2013. Balanced label propagation for partitioning massive graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*. ACM.
[35] Luis M Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. 2014. Adaptive partitioning for large-scale dynamic graphs. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*. IEEE, 144–153.
[36] Jim Webber. 2012. A Programmatic Introduction to Neo4j. In *Proceedings of the 3rd annual conference on Systems, Programming, and Applications: Software for Humanity*. ACM, 217–218.
[37] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *First International Workshop on Graph Data Management Experiences and Systems*.
[38] Yang Zhou, Ling Liu, Sangeetha Seshadri, and Lawrence Chiu. 2016. Analyzing enterprise storage workloads with graph modeling and clustering. *IEEE Journal on Selected Areas in Communications* 34, 3 (2016), 551–574.