Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud

Kisung Lee
Georgia Institute of Technology
kslee@gatech.edu

Ling Liu
Georgia Institute of Technology
lingliu@cc.gatech.edu

ABSTRACT

As the size and variety of information networks continue to grow in many scientific and engineering domains, we witness a growing demand for efficient processing of large heterogeneous graphs using a cluster of compute nodes in the Cloud. One open issue is how to effectively partition a large graph to process complex graph operations efficiently. In this paper, we present VB-Partitioner – a distributed data partitioning model and algorithms for efficient processing of graph operations over large-scale graphs in the Cloud. Our VB-Partitioner has three salient features. First, it introduces vertex blocks (VBs) and extended vertex blocks (EVBs) as the building blocks for semantic partitioning of large graphs. Second, VB-Partitioner utilizes vertex block grouping algorithms to place those vertex blocks that have high correlation in graph structure into the same partition. Third, VB-Partitioner employs a VB-partition guided query partitioning model to speed up the parallel processing of graph pattern queries by reducing the amount of inter-partition query processing. We conduct extensive experiments on several real-world graphs with millions of vertices and billions of edges. Our results show that VB-Partitioner significantly outperforms the popular random block-based data partitioner in terms of query latency and scalability over large-scale graphs.

Categories and Subject Descriptors

H.3.0 [**Information Storage and Retrieval**]: General

General Terms

Algorithms

Keywords

big data processing, heterogeneous graph, partitioning, cloud computing

SC '13 November 17-21, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503302

1. INTRODUCTION

Many real-world information networks consist of millions of vertices representing heterogeneous entities and billions of edges representing heterogeneous types of relationships among entities, such as Web-based networks, social networks, supply-chain networks and biological networks. One concrete example is the phylogenetic forests of bacteria, where each node represents a genetic strain of Mycobacterium tuberculosis complex (MTBC) and each edge represents a putative evolutionary change. Processing large heterogeneous graphs poses a number of unique characteristics in terms of big data processing. First, graph data is highly correlated and the topological structure of a big graph can be viewed as a correlation graph of its vertices and edges. Heterogeneous graphs add additional complexity compared to homogeneous graphs in terms of both storage and computation due to the heterogeneous types of entity vertices and entity links. Second, queries over graphs are typically subgraph matching operations. Thus, we argue that modeling heterogeneous graphs as a big table of entity vertices or entity links is ineffective for parallel processing of big graphs in terms of storage, network I/O and computation.

Hadoop MapReduce programming model and Hadoop Distributed File System (HDFS) are among the most popular distributed computing technologies for partitioning big data processing across a large cluster of compute nodes in the Cloud. HDFS (and its attached storage systems) is excellent for managing the big table data where row objects are independent and thus big data can be simply divided into equal-sized blocks (chunks) which can be stored and processed in parallel efficiently and reliably. However, HDFS is not optimized for storing and partitioning big datasets of high correlation, such as large graphs [17, 15]. This is because HDFS's block-based partitioning is equivalent to random partitioning of big graph data through either horizontal vertex-based partitioning or edge-based partitioning depending on whether the graph is stored physically by entity vertices or by entity links. Therefore, data partitions generated by such a random partitioning method tend to incur unnecessarily large inter-partition processing overheads due to the high correlation and thus the need for high degree of interactions among partitions in responding to a graph pattern query. Using such random partitioning method, even for simple graph pattern queries, the processing may incur unnecessarily large inter-partition join processing overheads due to the high correlation among partitions and demand multiple rounds of data shipping

across partitions hosted in multiple nodes of a compute cluster in the Cloud. Thus, Hadoop MapReduce alone is neither adequate for handling graph pattern queries over large graphs nor suitable for structure-based reasoning on large graphs, such as finding k-hop neighbors satisfying certain semantic constraints.

In this paper, we present a vertex block-based partitioning and grouping framework, called VB-Partitioner, for scalable and yet customizable graph partitioning and distributed processing of graph pattern queries over big graphs. VB-Partitioner supports three types of vertex blocks and a suite of vertex block grouping strategies, aiming at maximizing the amount of local graph processing and minimizing the network I/O overhead of inter-partition communication during each graph processing job. We demonstrate the efficiency and effectiveness of our VB-Partitioner by developing a VB-partition guided computation partitioning model that allows us to decompose graph pattern queries into desired vertex block partitions that are efficient for parallel query processing using a compute cluster.

This paper makes three novel contributions. First, we introduce vertex blocks and extended vertex blocks as the building blocks for partitioning a large graph. This vertex block-based approach provides a foundation for scalable and yet customizable data partitioning of large heterogeneous graphs by preserving the basic vertex structure. By scalable, we mean that data partitions generated by VB-Partitioner can support fast processing of big graph data of different size and complexity. By customizable, we mean that one partitioning technique may not fit all. Thus VB-Partitioner supports three types of vertex blocks and is by design adaptive to different data processing demands in terms of explicit and implicit structural correlations. Second, we develop a suite of vertex block grouping algorithms which enable efficient grouping of those vertex blocks that have high correlation in graph structure into one *VB* partition. We optimize the vertex block grouping quality by maximizing the amount of local graph processing and minimizing the inter-partition communication during each graph processing job. Third, to further utilize our vertex block-based graph partitioning approach, we introduce a *VB-partition* guided computation partitioning model, which allows us to transform graph pattern queries into vertex block-based graph query patterns. By partitioning and distributing big graph data using vertex block-based partitions, powered by the *VB-partition* guided query partitioning model, we can considerably reduce the inter-node communication overhead for complex query processing because most graph pattern queries can be evaluated locally on a partition server without requiring data shipping from other partition nodes. We evaluate our data partitioning framework and algorithms through extensive experiments using both benchmark and real datasets with millions of vertices and billions of edges. Our experimental results show that VB-Partitioner is scalable and customizable for partitioning and distributing big graph datasets of diverse size and structures, and effective for processing real-time graph pattern queries of different types and complexity.

2. OVERVIEW

2.1 Heterogeneous Graphs

We first define the heterogeneous graphs as follows.

Definition 1. (Heterogeneous Graph) Let $\mathcal{V}$ be a countably infinite set of vertex names, and Σ_V and Σ_E be a finite set of available types (or labels) for vertices and edges respectively. A heterogeneous graph is a directed, labeled graph, denoted as $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$ where V is a set of vertices (a finite subset of $\mathcal{V}$) and E is a set of directed edges (i.e., $E \subseteq V \times \Sigma_E \times V$). In other words, we represent each edge as a triple (v, l, v') which is a l-labeled edge from v to v'. l_V is a map from a vertex to its type ($l_V : V \rightarrow \Sigma_V$) and l_E is a map from an edge to its label ($l_E : E \rightarrow \Sigma_E$).

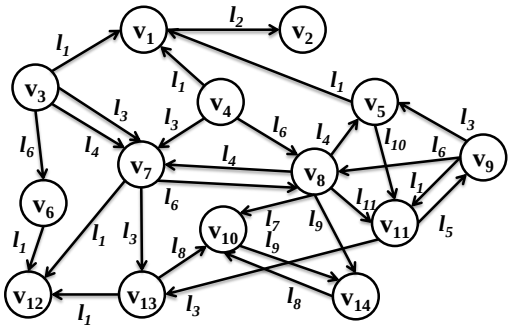


Figure 1: Heterogeneous graph

Fig. 1 shows an example of heterogeneous graphs. For example, there are several l_1-labeled edges such as (v_3, l_1, v_1), (v_4, l_1, v_1) and (v_{13}, l_1, v_{12}). Homogeneous graphs are special cases of heterogeneous graphs where vertices are of the same type, such as Web pages, and edges are of the same type, such as page links in a Web graph. In a heterogeneous graph, each vertex may have incoming edges (in-edges) and outgoing edges (out-edges). For example, in Fig. 1, vertex v_7 has 3 out-edges and 4 in-edge (i.e., 7 bi-edges).

Definition 2. (Out-edges, In-edges and Bi-edges) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the set of **out-edges** of a vertex $v \in V$ is denoted by $E_v^+ = \{(v, l, v') | (v, l, v') \in E\}$. Conversely, the set of **in-edges** of v is denoted by $E_v^- = \{(v', l, v) | (v', l, v) \in E\}$. We also define **bi-edges** of v as the union of its **out-edges** and **in-edges**, denoted by $E_v^\pm = E_v^+ \cup E_v^-$.

Definition 3. (Path) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, an **out-edge path** from a vertex $u \in V$ to another vertex $w \in V$ is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $v_0 = u$, $v_k = w$, $\forall m \in [0, k-1] : (v_m, l_m, v_{m+1}) \in E$. Conversely, an **in-edge path** from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $u = v_0$, $w = v_k$, $\forall m \in [0, k-1] : (v_{m+1}, l_m, v_m) \in E$. A **bi-edge path** from vertex u to vertex w is a sequence of vertices, denoted by $v_0, v_1, \ldots, v_k$, such that $u = v_0$, $w = v_k$, $\forall m \in [0, k-1] : (v_m, l_m, v_{m+1}) \in E$ or $(v_{m+1}, l_m, v_m) \in E$. The **length** of the path $v_0, v_1, \ldots, v_k$ is k.

Definition 4. (Hop count) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the **out-edge hop count** from a vertex $u \in V$ to another vertex $w \in V$, denoted by $hop^+(u, w)$, is the minimum length of all possible *out-edge* paths from u to w. We also define the out-edge hop count from u to an out-edge (w, l, w') of w, denoted by $hop^+(u, wlw')$, as $hop^+(u, w) + 1$. The hop count $hop^+(u, w)$ is zero if $u = w$ and ∞ if there is no out-edge path from u to w.

The in-edge and bi-edge hop counts are similarly defined using the in-edge and bi-edge paths respectively.

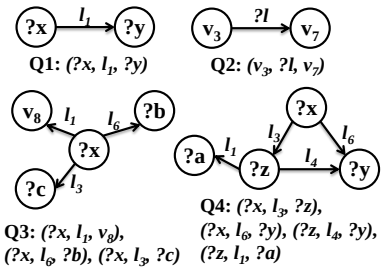


Figure 2: Graph pattern query graphs

2.2 Operations on Heterogeneous Graphs

Graph pattern queries [7] are subgraph matching problems and are widely recognized as one of the most fundamental graph operations. A graph pattern is often expressed in terms of a set of vertices and edges such that some of them are variables. Processing of a graph pattern query is to find a set of vertex or edge values on the input graph which can be substituted for the variables while satisfying the structure of the graph pattern. Therefore, processing a graph pattern query can be viewed as solving a subgraph matching problem or finding missing vertex or edge instantiation values in the input graph.

A *basic graph pattern* is an edge (v, l, v') in which any combination of the three elements can be variables. We represent variables with a prefix "?" such as `?x` to differentiate variables from the instantiation of vertex names and edge labels.

A *graph pattern* consists of a set of basic graph patterns. If there is a variable shared by several basic graph patterns, the returned values for the variable should satisfy all the basic graph patterns which include the variable. For example, a graph pattern $\{(?x, l_1, v_8),\ (?x, l_6, ?a),\ (?x, l_3, ?b)\}$ requests those vertices that have l_1-labeled out-edge to v_8 and also l_6-labeled and l_3-labeled out-edges. It also requests the connected vertices (i.e., $?a$ and $?b$) linked by the out-edges. This type of operations is very common in social networks when we request additional information of users satisfying a certain condition such as $\{(?member, affiliation, GT),\ (?member, hometown, ?city),\ (?member, birthday, ?date)\}$. Another graph pattern $\{(?x, l_3, ?z),\ (?x, l_6, ?y),\ (?z, l_4, ?y),\ (?z, l_1, ?a)\}$ requests all vertices such that each vertex x has any l_3-labeled (to z) and l_6-labeled (to y) out-edges and there is any l_4-labeled edge from z to y and z has any l_1-labeled out-edge. This type of operations is also common in social networks when we want to find friends of friends within k-hops satisfying a certain condition. We formally define the graph pattern as follows.

Definition 5. (graph pattern) Let $\mathcal{V}_{var}$ and $\mathcal{E}_{var}$ be countably infinite sets of vertex variables and edge variables respectively. Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, a graph pattern is $G_q = (V_q, E_q, \Sigma_{V_q}, \Sigma_{E_q}, l_{V_q}, l_{E_q})$ where $V_q \subseteq V \cup \mathcal{V}_{var}$ and $E_q \subseteq V_q \times (\Sigma_E \cup \mathcal{E}_{var}) \times V_q$.

For example, $\{(?member, work, ?company),\ (?member, friend, ?friend),\ (?friend, work, ?company),\ (?friend, friend, ?friend2)\}$ requests, for each user, friends of her friends who are working in the same company with her. Fig. 2 gives four typical graph pattern queries (selection by edge, selection by vertices, star join and complex join).

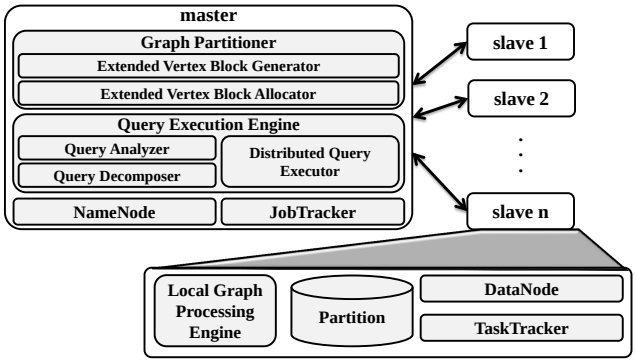


Figure 3: System Architecture

2.3 System Architecture

The first prototype of our VB-Partitioner framework is implemented on top of a Hadoop cluster. We use Hadoop MapReduce and HDFS to partition heterogeneous graphs and manage distributed query execution across a cluster of Hadoop nodes in the Cloud. Fig. 3 shows a sketch of the system architecture. Our system consists of one *master* node and a set of *slave* nodes. When we execute a graph partitioning or distributed query processing algorithm using Hadoop, the master node serves as the NameNode of HDFS and the JobTracker of Hadoop MapReduce. Similarly, the slave nodes serve as the DataNodes of HDFS and the TaskTrackers of Hadoop MapReduce.

Graph Partitioner. Many real-world big graphs exceed the performance capacity (e.g., memory, CPU) of a single node. Thus, we provide a distributed implementation of our VB-Partitioner on a Hadoop cluster of compute nodes. Concretely, we first load the big input graph into HDFS and thus the input graph is split into large HDFS chunks and stored in a cluster of slave nodes. **Extended vertex block generator** generates vertex block or extended vertex block for each vertex in the input graph stored in HDFS using Hadoop MapReduce. **Extended vertex block allocator** performs two tasks to place each vertex block to a slave node of the Hadoop cluster: (i) It employs a vertex block grouping algorithm to assign each extended vertex block to a partition; (ii) It assigns each partition to a slave node, for example using a standard hash function, which will balance the load by attempting to assign equal number of partitions to each slave node. On each slave node, a local graph processing engine is installed to process graph pattern queries against the partitions locally stored on the node. We provide more detail on our graph partitioning algorithms in Section 3.

Query Execution Engine. To speed up the processing of graph pattern queries, we first categorize our distributed query execution into two types: *intra*-partition processing and *inter*-partition processing. By *intra*-partition processing, we mean that a graph query Q can be fully executed in parallel on each slave node without any cross-node coordination. The only communication cost required to process Q is for the master node to dispatch Q to each slave node. If no global sort of results is required, each slave node can directly (or via its master to) return its locally generated results. Otherwise, either the master node or an elected slave node will be served as the integrator node to merge the partial results received from all slave nodes to generate the final

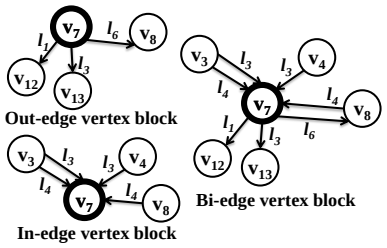


Figure 4: Different vertex blocks of v_7

sorted results of Q. By *inter*-partition processing, we mean that a graph query Q as a whole cannot be executed on any slave node, and thus it needs to be decomposed into a set of subqueries such that each subquery can be evaluated by intra-partition processing. Thus, the processing of Q requires multiple rounds of coordination and data transfer across a set of slave nodes. In contrast to intra-partition processing, the network I/O (communication) cost can be extremely high, especially when the number of subqueries is not small and the size of intermediate results to be transferred across the cluster of slave nodes is large.

For a given graph query Q, **query analyzer** analyzes Q to see whether Q can be executed using intra-partition processing. If Q can be executed using intra-partition processing, Q is directly sent to distributed query executor. Otherwise, **query decomposer** is invoked to split Q into a set of subqueries such that each subquery can be executed using intra-partition processing. **Distributed query executor** is in charge of executing Q using intra-partition or inter-partition processing by coordinating slave nodes. We will explain our distributed query processing in detail in Section 4.

3. VB-PARTITIONER: DESIGN FRAMEWORK

The VB-Partitioner framework for heterogeneous graphs consists of three phases. First, we build a vertex block for each vertex in the graph. We guarantee that all the information (vertices and edges) included in a vertex block will be stored in the same partition and thus on the same slave node. Second, for a section of vertices, we expand their vertex blocks (VBs) to the extended vertex block (EVBs). Third but not the least, we employ a VB grouping algorithm to assign each VB or EVB to a vertex block-based partition. We below describe each of the three phases in detail.

3.1 Vertex Blocks

A vertex block consists of an anchor vertex and its connected edges and vertices. To support customizable and effective data partitioning, we introduce three different vertex blocks based on the direction of connected edges of the anchor vertex: 1) out-edge vertex block 2) in-edge vertex block and 3) bi-edge vertex block. Fig. 4 shows out-edge, in-edge and bi-edge vertex blocks of vertex v_7 in Fig. 1 respectively. We formally define the concept of vertex block as follows.

Definition 6. (Vertex block) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, **out-edge vertex block** of an anchor vertex $v \in V$ is a subgraph of G which consists of v and all its out-edges, denoted by $VB_v^+ = (V_v^+, E_v^+, \Sigma_{V_v^+}, \Sigma_{E_v^+}, l_{V_v^+}, l_{E_v^+})$ such that $V_v^+ = \{v\} \cup \{v^+ | v^+ \in V, (v, l, v^+) \in E_v^+\}$. Similarly, **in-edge vertex block** of v is defined as $VB_v^- = (V_v^-, E_v^-, \Sigma_{V_v^-}, \Sigma_{E_v^-}, l_{V_v^-}, l_{E_v^-})$ such that $V_v^- = \{v\} \cup \{v^- | v^- \in V, (v^-, l, v) \in E_v^-\}$. Also, **bi-edge vertex block** of v is defined as $VB_v^\pm = (V_v^\pm, E_v^\pm, \Sigma_{V_v^\pm}, \Sigma_{E_v^\pm}, l_{V_v^\pm}, l_{E_v^\pm})$ such that $V_v^\pm = \{v\} \cup \{v^\pm | v^\pm \in V, (v, l, v^\pm) \in E_v^+ \text{ or } (v^\pm, l, v) \in E_v^-\}$.

Each vertex block preserves the basic graph structure of a vertex and thus can be used as an atomic unit (building block) for graph partitioning. By placing a vertex block into the same partition, we can efficiently process all *basic* graph pattern queries using intra-partition processing, such as selection by edge or by vertex, because it guarantees that all vertices and edges required to evaluate such queries are located in the same partition. Consider the graph pattern query $Q2$ $(v_3, ?l, v_7)$ in Fig. 2. We can process the query using intra-partition processing regardless of the type of the vertex block. If we use *out-edge* (or *in-edge*) vertex blocks for partitioning, it is guaranteed that all out-edges (or in-edges) of v_3 (or v_7) are located in the same partition. It is obviously true for *bi-edge* vertex blocks because it is the union of *in-edge* and *out-edge* vertex blocks.

It is worth noting that each partitioning scheme based on each of the three types of vertex blocks can be advantageous for some queries but fail to produce the results of queries effectively. Consider $Q3$ $\{(?x, l_1, v_8), (?x, l_6, ?a), (?x, l_3, ?b)\}$ in Fig. 2. It is guaranteed that all out-edges of any vertex matching `?x` are located in the same partition if we use *out-edge* vertex blocks. This enables the query evaluation using intra-partition processing because only out-edges of `?x` are required. However, if we use *in-edge* vertex blocks, we can no longer evaluate $Q3$ solely using intra-partition processing because we can no longer guarantee that all out-edges of any vertex matching `?x` are located in the same partition.

Consider $Q4$ $\{(?x, l_3, ?z), (?x, l_6, ?y), (?z, l_4, ?y), (?z, l_1, ?a)\}$ in Fig. 2. We cannot process $Q4$ using intra-partition processing because there is no vertex (or vertex variable) which can cover all edges in the query graph using its out-edges, in-edges or even bi-edges. For example, if we consider *bi-edge* vertex block of `?z`, it is clear that there is one remaining edge $((?x, l_6, ?y))$ which cannot be covered by the vertex block. This motivates us to introduce the concept of extended vertex block.

3.2 Extended Vertex Blocks

The basic idea of the extended vertex block is to include not only directly connected edges of the anchor vertex but also those within k-hop distance from the anchor vertex. Concretely, to construct the extended vertex block of an anchor vertex, we extend its vertex block hop by hop to include those edges (and their vertices) that are reachable within k hops from the anchor vertex. For example, from the out-edge vertex block of v_7 in Fig. 4, its 2-hop (k=2) extended vertex block will add the out-edges of v_8, v_{12} and v_{13}.

One of the most significant advantages of k-hop extended vertex blocks is that most graph pattern queries can be executed using intra-partition processing without any coordination with another partition. However, when k is too large relative to the size of the graph, extended vertex blocks can be costly in terms of the storage cost on each node. In other words, even though we remove inter-partition communication cost, the slow local processing on each large partition may become the dominating factor for the query processing.

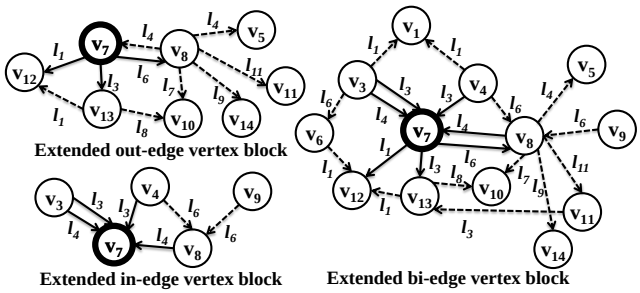


Figure 5: 2-hop extended vertex blocks of v_7

To tackle this problem, we introduce a *k-hop* extended vertex block in which the extension level is controlled by the system parameter k. As a base case, the *1-hop* extended vertex block of an anchor vertex is the same as its vertex block. The *k-hop* extended vertex block of an anchor vertex includes all vertices and edges in its *(k-1)-hop* extended vertex block and additional edges (and their vertices) which are connected to any vertex in the *(k-1)-hop* extended vertex block.

We also define three different types of the k-hop extended vertex block based on the direction of expanded edges: 1) k-hop extended out-edge vertex block; 2) k-hop extended in-edge vertex block and 3) k-hop extended bi-edge vertex block. Fig. 5 shows the different types of 2-hop extended vertex block for v_7. Dotted edges indicate the newly added edges from the corresponding vertex block.

3.3 VB-based Grouping Techniques

After we obtain a vertex block or an extended vertex block of each vertex in the input graph, we enter the second phase of VB-Partitioner. It strategically groups a subset of VBs and EVBs into a VB-partition by employing our vertex block-based grouping algorithms such that highly correlated VBs and EVBs will be placed into one VB-partition. We remove any duplicate vertices and edges within each VB-partition.

When assigning each VB or EVB to a partition, we need to consider the following three factors for generating efficient and effective partitions: (i) The generated partitions should be well balanced; (ii) The amount of replications should be small; and (iii) The formation of VB-partitions should be fast and scalable. First, balanced partitions are important for efficient query processing because one big partition, in the imbalanced partitions, can be a bottleneck and increase the overall query processing cost. Second, we need to reduce the number of replicated vertices and edges to construct smaller partitions and thus support faster local query processing in each partition. Since an edge (and its vertices) can be included in several extended vertex blocks, we need to assign those extended vertex blocks sharing many edges to the same partition to reduce the number of replicated edges and vertices. Third but not the least, we need to support fast partitioning for frequently updated graphs. Since one partitioning technique cannot fit all, we propose three different grouping techniques in which each has its own strength and thus can be accordingly selected for different graphs and query types.

Hashing-based VB Grouping. The hashing-based grouping technique assigns each extended vertex block based on the hash value of the block's anchor vertex name. This partitioning technique generates well-balanced partitions and is very fast. However, the hashing-based VB grouping is not effective in terms of managing and reducing the amount of vertex and edge replication because the hashing-based algorithm pays no attention on the correlation among different VBs and EVBs. If we can develop a smart hash function that is capable of incorporating some domain knowledge about vertex names, we can reduce the number of replicated edges. For example, if we know that vertices sharing the same prefix (or suffix) in their name are closely connected in the input graph, we can develop a new hash function, which uses only the prefix (or suffix) of the vertex names to calculate the hash values, and assign the vertices sharing the common prefix (or suffix) to the same partition.

Minimum cut-based VB Grouping. The minimum cut-based grouping technique utilizes the minimum cut graph partitioning algorithm, which splits an input graph into smaller components by minimizing the number of edges running between the components. After we run the graph partitioning algorithm for an input graph by setting the number of components as the number of partitions, we can get a list which has the assigned component id for each vertex. Since the algorithm assigns each vertex to one component and there is an one-to-one mapping between components and partitions, we can directly utilize the list of components by assigning each VB or EVB to the partition corresponding to the assigned component of its anchor vertex. This grouping technique is very good for reducing the number of replicated edges because we can view the minimum cut algorithm as grouping closely located (or connected) vertices in the same component. Also, because another property of the minimum cut algorithm is to generate uniform components such that the components are of about the same size, this grouping technique can also achieve a good level of balanced partitions. However, the uniform graph partitioning problem is known to be NP-complete [6]. It often requires a long running time for VB-grouping due to its high time complexity. Our experiments on large graphs in Section 5 show that the minimum cut-based VB grouping is practically infeasible for large and complex graphs.

High degree vertex-based VB Grouping. This grouping approach is motivated for providing a better balance between reducing replication and fast processing. The basic idea of this grouping algorithm is to find some high degree vertices with many in-edges and/or out-edges and place the VBs or EVBs of those nearby vertices of each high degree vertex in the same partition of the high degree vertex. By focusing on only high degree vertices, we can effectively reduce the time complexity of grouping algorithm and better control the degree of replications.

Concretely, we first find some high degree vertices whose number of connected edges is larger than a system-supplied threshold value δ. If we increase the δ value, a smaller number of vertices would be selected as the high degree vertices.

Second, for each high degree vertex, we find a set of vertices, called *dependent* vertices, which are connected to the high degree vertex by one hop. There are three types of dependent vertices for each high degree vertex (out-edge, in-edge or bi-edge). If the high degree vertex has an *out-edge* EVB, then we find its dependent vertices by following the *in-edges* of the high degree vertex. Similarly, we check

the *out-edges* and *bi-edges* of the high degree vertex for extended *in-edge* and *bi-edge* vertex blocks respectively.

Third, we group each high degree vertex and its dependent vertices to assign them (and their extended vertex blocks) to the same partition. If a vertex is a dependent vertex of multiple high degree vertices, we merge all its high degree vertices and their dependent vertices in the same group. By doing so, we can prevent the replication of the high degree vertices under 2-hop extended *out-edge* vertex blocks. If 3-hop extended *out-edge* vertex blocks are generated, we also extend the dependent vertex set of a high degree vertex by including additional vertices which are connected to any dependent vertex by one hop. We can repeatedly extend the dependent vertex set for $k > 3$. To prevent from generating a huge partition, we exclude those groups, whose size (the number of vertices in the group) is larger than a threshold value, when we merge groups. By default, we divide the number of all vertices in the input graph by the number of partitions and use the result as the threshold value to identify such huge partitions.

Finally, we assign the extended vertex blocks of all vertices in a high-degree group to the same partition. For each uncovered vertex which is not close to any high degree vertex, we simply select a partition having the smallest size and assign its extended vertex block to that partition.

4. DISTRIBUTED QUERY PROCESSING

For a given graph pattern query Q, the first step is to analyze Q to determine whether Q can be executed using intra-partition processing or not. If yes, Q is directly sent to the query execution step without invoking the query decomposition step. Otherwise, we iteratively decompose Q into a set of subqueries such that each subquery can be executed using intra-partition processing. Finally, we generate execution plans for Q (intra-partition processing) or for its subqueries (inter-partition processing) and the query result by executing the plans using the cluster of compute nodes.

4.1 Query Analysis

In query analysis step, we need to determine whether a query Q needs to be sent to the query decomposer or not. The decision is primarily based on eccentricity, radius and center vertex in the context of graph.

Definition 7. (Eccentricity) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the out-edge **eccentricity** ϵ^+ of a vertex $v \in V$ is the greatest out-edge hop count from v to any edge in G and formally defined as follows:

$$\epsilon^+(v) = \max_{(w,l,w') \in E} hop^+(v, wlw')$$

The in-edge eccentricity ϵ^- and bi-edge eccentricity $\epsilon^\pm$ are similarly defined. The eccentricity of a vertex in a graph can be thought of as how far a vertex is from the vertex most distant from it in the graph.

Definition 8. (Radius and Center vertex) Given a graph $G = (V, E, \Sigma_V, \Sigma_E, l_V, l_E)$, the out-edge **radius** of G, denoted by $r^+(G)$, is the minimum out-edge eccentricity of any vertex $v \in V$ and formally defined as follows:

$$r^+(G) = \min_{v \in V} \epsilon^+(v)$$

The out-edge **center vertices** of G, denoted by $CV^+(G)$, are the vertices whose out-edge eccentricity equals to the out-edge radius of G and formally:

$$CV^+(G) = \{v | v \in V, \epsilon^+(v) = r^+(G)\}$$

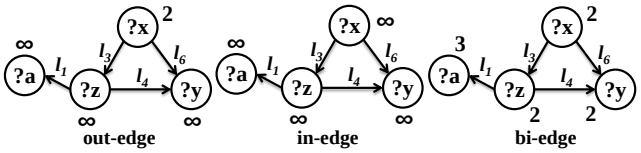


Figure 6: Query Analysis

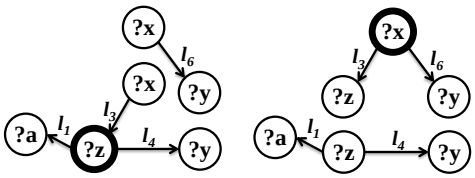
The in-edge radius $r^-(G)$, in-edge center vertices $CV^-(G)$, bi-edge radius $r^\pm(G)$ and bi-edge center vertices $CV^\pm(G)$ are similarly defined.

Assuming that the partitions are constructed using k-hop extended vertex blocks, for a graph pattern query Q and its query graph G_Q, we first calculate the *radius* and the center vertices of the query graph based on Definition 8. If the partitions are constructed using extended out-edge (in-edge or bi-edge) vertex blocks, we calculate $r^+(G_Q)$ ($r^-(G_Q)$ or $r^\pm(G_Q)$) and $CV^+(G_Q)$ ($CV^-(G_Q)$ or $CV^\pm(G_Q)$). If the radius is equal to or less than k, then the query Q as a whole can be executed using the intra-partition processing. This is because, from the center vertices of G_Q, our k-hop extended vertex blocks guarantee that all edges that are required to evaluate Q are located in the same partition. In other words, by choosing one of the center vertices as an anchor vertex, it is guaranteed that the k-hop extended vertex block of the anchor vertex covers all the edges in G_Q given that the radius of G_Q is not larger than k. Therefore we can execute Q without any coordination and data transfer among the partitions. If the radius is larger than k, we need to decompose Q into a set of subqueries.

Fig. 6 presents how our query analysis step works under three different types (out-edge, in-edge and bi-edge) of extended vertex blocks for graph pattern query $Q4$ in Fig. 2. The eccentricity value of each vertex is given next to the vertex. Since the **out-edge** radius of the query graph is 2, we can execute the query using intra-partition processing if the partitions are constructed using k-hop extended *out-edge* vertex blocks and k is equal to or larger than 2. However, the **in-edge** radius of the query graph is *infinity* because there is no vertex which has at least one in-edge path to all the other vertices. Therefore, we cannot execute the query using intra-partition processing if the partitions are constructed using extended *in-edge* vertex blocks.

4.2 Query Decomposition

To execute a graph pattern query Q using *inter-partition* processing, it is necessary to slit Q into a set of subqueries in which each subquery can be executed using intra-partition processing. Given that using Hadoop and HDFS to join the partial results generated from the subqueries, we need to carefully decompose Q in order to minimize the join processing cost. Since we use one Hadoop job to join two sets of partial results and each Hadoop job has an initialization overhead of about 10 seconds regardless of the input data size, we decompose Q by minimizing the number of subqueries. To find such decomposition, we use an intuitive approach which first checks whether Q can be decomposed into *two* subqueries such that each subquery can be evaluated using intra-partition processing. To check whether a



(a) from ?z (b) from ?x

Figure 7: Query decomposition (bi-edge)

subquery can be executed using intra-partition processing, we calculate the radius of the subquery's graph and then perform the query analysis steps outlined in the previous section. We repeat this process until at least one satisfying decomposition is found.

Concretely, we start the query decomposition by putting all vertices in the query graph G_Q of Q into a set of candidate vertices to be examined in order to find such a decomposition having two subqueries. For each candidate vertex v, we draw the k-hop extended vertex block of v in G_Q, assuming that the partitions are constructed using k-hop extended vertex blocks. For the remaining edges of G_Q, which are not covered by the k-hop extended vertex block of v, we check whether there is any other candidate vertex whose k-hop extended vertex block in G_Q can fully cover the remaining edges. If there is such a decomposition, we treat each subgraph as a subquery of Q. Otherwise, we increase the number of subqueries by one and then repeat the above process until we find a satisfying decomposition. If we find more than one satisfying decompositions having the equal number of subqueries, then we choose the one in which the standard deviation of the size (i.e., the number of edges) of subqueries is the smallest, under the assumption that a small subquery may generate large intermediate results. We leave as future work the query optimization problem where we can utilize additional metadata such as query selectivity information.

For example, let us assume that the partitions are constructed using 1-hop extended *bi-edge* vertex blocks and thus graph pattern query $Q4$ in Fig. 2 cannot be executed using intra-partition processing. To decompose the query, if we start with vertex `?z`, we will get a decomposition which consists of two subqueries as shown in Fig.7(a). If we start with vertex `?x`, we will also get two subqueries as shown in Fig.7(b). Based on the smallest subquery standard deviation criterion outlined above, we choose the latter because two subqueries are of the same size.

5. EXPERIMENTAL EVALUATION

In this section, we report the experimental evaluation results of our partitioning framework for various heterogeneous graphs. We first explain the characteristics of datasets we used for our evaluation and the experimental settings. We divide the experimental results into four categories: (i) We show the **data partitioning and loading time** for different extended vertex blocks and grouping techniques and compare it with the data loading time in a single server. (ii) We present the **balance and replication level** of generated partitions using different extended vertex blocks and grouping techniques. (iii) We conduct the experiments on **query processing latency** using various types of graph pattern queries. (iv) We also evaluate the scalability of our partitioning framework by increasing the dataset size and the number of servers in the cluster.

5.1 Datasets

To show the working of our partitioning framework for various graphs having totally different characteristics, we not only use three real graphs but also generate three graphs from each of two different benchmark generators. As real graphs, we choose **DBLP** [1] containing bibliographic information in computer science, **Freebase** [2] which is a large knowledge base and **DBpedia** [4] having structured information from Wikipedia. As benchmark graphs, we choose LUBM and SP^{2}Bench, which are widely used for evaluating RDF storage systems, and generate **LUBM2000**, **LUBM4000**, **LUBM8000** using LUBM and **SP2B-100M**, **SP2B-200M** and **SP2B-500M** using SP^{2}Bench. As a data cleaning step, we remove any duplicate edges using one Hadoop job for each dataset. Table 1 shows the number of vertices and edges and the average number of out-edges and in-edges of the datasets. Note that the benchmark datasets, generated from the same benchmark generator, have almost the same average number of out-edges and in-edges regardless of the dataset size. Fig. 8 shows the out-edge and in-edge distribution of the datasets. In the x-axis of the figures, we plot the number of out-edges (or in-edges) and in the y-axis we plot the percentage of vertices whose number of out-edges (or in-edges) is equal to or less than this number of out-edges (or in-edges). For example, about 85%, 97% and 89% of vertices have 25 or less out-edges on DBLP, Freebase and DBpedia respectively. Note that the benchmark datasets, generated from the same benchmark generator, have almost the same out-edge and in-edge distribution regardless of the dataset size. We omit the results of LUBM8000 and SP2B-500M because each has almost the same distribution with datasets from the same benchmark generator.

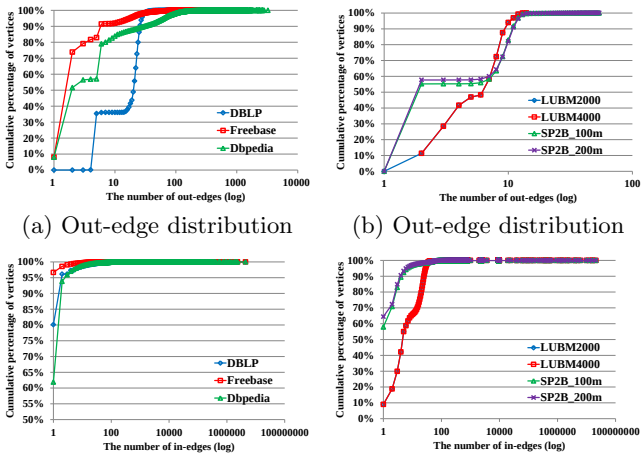
Table 1: Datasets

Dataset	#vertices	#edges	avg. #out	avg. #in
DBLP	25,901,515	56,704,672	16.66	2.39
Freebase	51,295,293	100,692,511	4.41	2.11
DBpedia	104,351,705	287,957,640	11.62	2.82
LUBM2000	65,724,613	266,947,598	6.15	8.27
LUBM4000	131,484,665	534,043,573	6.15	8.27
LUBM8000	262,973,129	1,068,074,675	6.15	8.27
SP2B-100M	55,182,878	100,000,380	5.61	2.11
SP2B-200M	111,027,855	200,000,007	5.49	2.08
SP2B-500M	280,908,393	500,000,912	5.31	2.04

5.2 Setup

We use a cluster of 21 nodes (one is the master node) on Emulab [20]: each has 12 GB RAM, one 2.4 GHz 64-bit quad core Xeon E5530 processor and two 7200 rpm SATA disks (250GB and 500GB). The network bandwidth is about 40 MB/s. When we measure the query processing time, we perform five cold runs under the same setting and show the *fastest* time to remove any possible bias posed by OS and/or network activity.

As a local graph processing engine, we install RDF-3X version 0.3.5 [18], on each slave server, which an open-source RDF management system. We use Hadoop version 1.0.4 running on Java 1.6.0 to run our graph partitioning algo-



(a) Out-edge distribution (b) Out-edge distribution

(c) In-edge distribution (d) In-edge distribution

Figure 8: Out-edge and In-edge Distribution

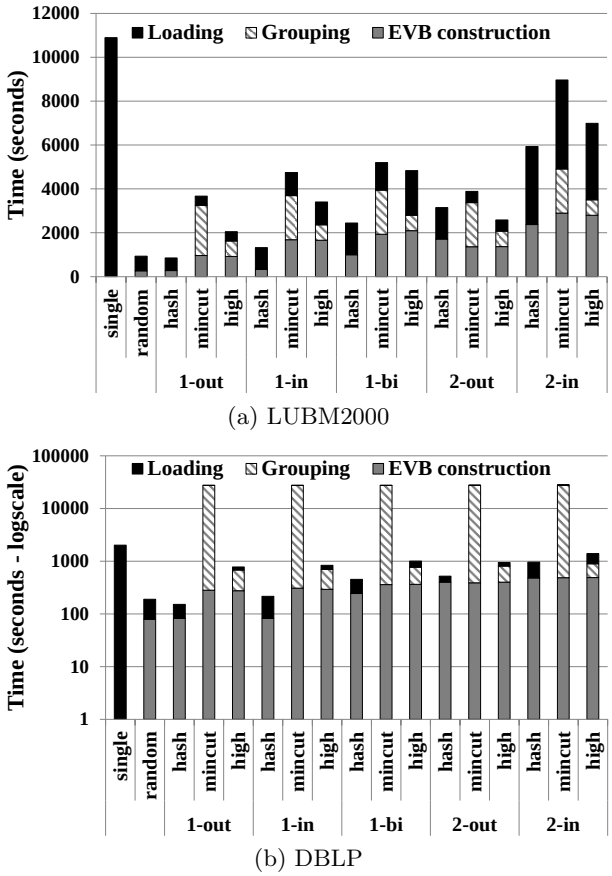
rithms and join the intermediate results, generated by subqueries, during inter-partition processing. For comparison, we also implement random partitioning. To implement the minimum-cut based VB grouping technique, we use graph partitioner METIS version 5.0.2 [5] with its default configuration.

To simplify the name of our extended vertex blocks and grouping techniques, we use `[k]-[out|in|bi]-[hash|mincut|high]` as our naming convention. For example, *1-out-high* indicates the high degree vertex-based technique with 1-hop extended out-edge vertex blocks.

5.3 Partitioning and Loading Time

We first compare the partitioning and loading time of our framework with that on a single server. Fig. 9 shows the partitioning and loading time of LUBM2000 and DBLP for different extended vertex blocks and grouping techniques. The loading time indicates the loading time of RDF-3X. The single server approach has only the loading time because there is no partitioning. To support efficient partitioning, we implement the extended vertex block construction and grouping using Hadoop MapReduce in the cluster of nodes. Since the hashing-based grouping technique simply uses a hash function (By default, we use the hash function of Java String class) to assign each extended vertex block to a partition, we incorporate the grouping step into the construction step and thus there is no grouping time for those using the hashing-based grouping technique. The grouping time of the minimum cut-based grouping technique includes both the input conversion time (from RDF to METIS input format) and METIS running time. We also implement the input conversion step using Hadoop MapReduce in the cluster of nodes for efficient conversion.

Fig. 9(a) clearly shows that we can significantly reduce the graph loading time by using our partitioning framework, compared to using only single server. The only exception is when we use the minimum cut-based grouping technique in which we need to convert the datasets into the METIS input formats, as shown in Fig. 9(b). The conversion time depends on not only the dataset size but also the structure of the graph. For example, the conversion times of DBLP and Freebase are about 7.5 hours and 35 hours respectively,



(a) LUBM2000

(b) DBLP

Figure 9: Partitioning and Loading Time

which are much longer than 50 minutes of DBpedia even though DBpedia has much more edges. We think that this is because DBLP and Freebase include some vertices having a huge number of connected edges. For example, there are 4 and 6 vertices having more than one million in-edges on DBLP and Freebase respectively. Also note that the minimum cut-based grouping technique couldn't work on LUBM4000, LUBM8000 and SP2B-500M because METIS failed due to the insufficient memory on a single machine with 12 GB RAM. This result indicates that the minimum cut-based grouping technique is infeasible for some graphs having a huge number of vertices and edges and/or complex structure.

5.4 Balance and Replication level

To show the balance of generated partitions in terms of the number of edges, we use the *relative standard deviation expressed as a percentage*, defined as the ratio of the standard deviation to the mean (and then multiplied by 100 to be expressed as a percentage). A higher percentage means that the generated partitions are less balanced. Fig. 10 shows the relative standard deviation for different extended vertex blocks and grouping techniques. As we expect, the hashing-based grouping technique generates the most balanced partitions for most cases. Especially, using extended *out-edge* vertex blocks, the hashing-based technique constructs almost perfectly balanced partitions. It is interesting to note that, the hashing-based technique using

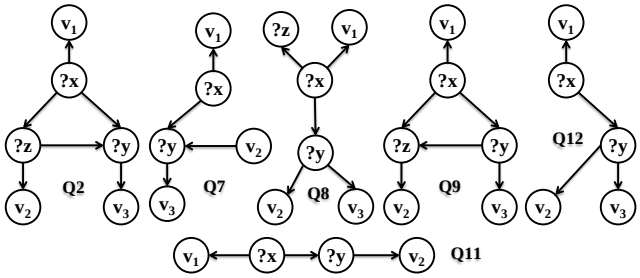


Figure 12: Benchmark query graphs

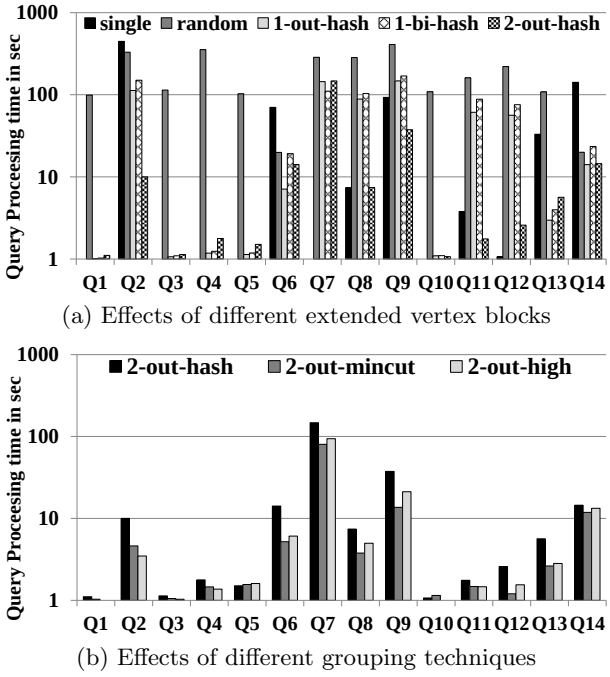
extended in-edge vertex blocks generates less balanced partitions than that using out-edge EVBs. This is because there are some vertices having a huge number of in-edges (e.g., more than one million in-edges) as shown in Fig. 8(c) and Fig. 8(d). Therefore, partitions including the extended vertex blocks of such vertices will have much more edges than the others. We omit the results of LUBM4000, LUBM8000, SP2B-500M and SP2B-500M because each has almost the same relative standard deviation with the dataset from the same benchmark generator.

To see how many edges are replicated, Fig. 11 shows the total number of edges of all the generated partitions for different extended vertex blocks and grouping techniques. As we expect, the minimum cut-based grouping technique is the best in terms of reducing the replication. Especially, when we use 2-hop out-edge EVBs, the minimum cut-based grouping technique replicates only a small number of edges. However, for the other vertex blocks, the benefit of the minimum cut-based grouping technique is not so significant if we consider its overhead as shown in Fig. 9. Also recall that the minimum cut-based grouping technique fails to work on LUBM4000, LUBM8000 and SP2B-500M because METIS failed due to the insufficient memory.

5.5 Query Processing

Since LUBM provides 14 benchmark queries, we utilize them to evaluate query processing in the partitions generated by our partitioning framework. Among 14 queries, two queries (Q6 and Q14) are basic graph pattern queries (i.e., only one edge in their query graph) and 6 queries (Q1, Q3, Q4, Q5, Q10 and Q13) are star-like queries in which all the edges in their query graph are out-edges from one vertex variable. Fig. 12 shows the graph pattern query graphs for the other queries. We omit the edge label because there is no edge variable.

Fig. 13 shows the query processing time of all 14 benchmark queries for different extended vertex blocks and grouping techniques on LUBM2000. For brevity, we omit the results of using 1-hop and 2-hop extended *in-edge* vertex blocks because they are not adequate for the benchmark queries due to many leaf-like vertices which have only one in-edge and no out-edge. All partitioning approaches using 1-hop out-edge EVBs, 1-hop bi-edge EVBs and 2-hop out-edge EVBs ensure intra-partition processing for the basic graph pattern queries (Q6 and Q14) and star-like queries (Q1, Q3, Q4, Q5, Q10 and Q13). Among the remaining queries (Q2, Q7, Q8, Q9, Q11 and Q12), no query can be executed using intra-partition processing over 1-hop extended out-edge and bi-edge vertex blocks. On the other hand, 2-hop out-edge



(a) Effects of different extended vertex blocks

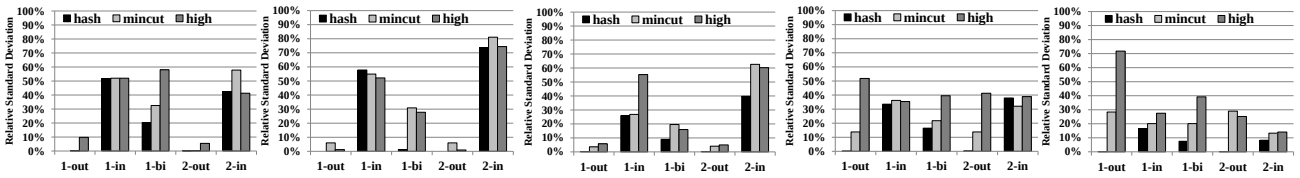
(b) Effects of different grouping techniques

Figure 13: Query Processing Time on LUBM2000

EVBs guarantee intra-partition processing for all the benchmark queries except Q7 in which 2-hop extended out-edge vertex block of `?x` cannot cover the edge from v_2 to `?y`.

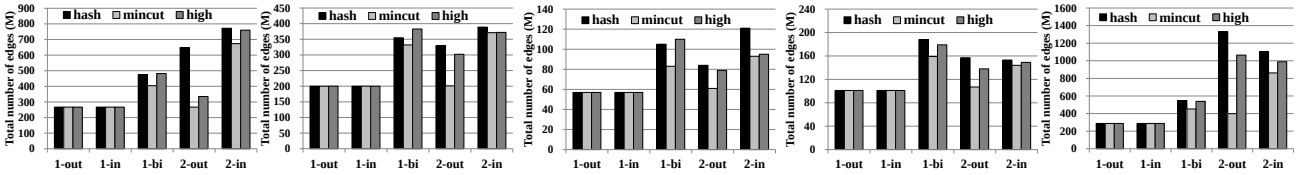
The result clearly shows the huge benefit of intra-partition processing, compared to inter-partition processing. For example, for Q2, the query processing time over 2-hop out-edge EVBs is only 4% of that over 1-hop out-edge EVBs as shown in Fig. 13(a). That is two orders of magnitude faster than the result on a single server. If we use inter-partition processing, it is much slower than using intra-partition processing due to the initialization overhead of Hadoop and large size of intermediate results. For example, the size of the intermediate results for Q7 over 2-hop out-edge EVBs is 1.2 GB which is much larger than the final result size of 907 bytes. The result for Q7 also shows the importance of the number of subqueires in inter-partition processing. The query processing over 2-hop out-edge EVBs, which consists of 2 subqueries, is only 65% of that over 1-hop out-edge EVBs, which consists of 3 subqueries, even though the partitions generated using 2-hop out-edge EVBs are much larger as shown in Fig. 11(a). For star query Q1, Q3, Q4, Q5 and Q10 having very high selectivity (i.e., the result size is less than 10kb), the query processing is usually fast (less than 2 seconds) in the partitions generated by our framework. However, it is slight slower than the query processing on a single server because there is some overhead on the master node which sends the query to all the slave nodes and merges the partial results received from the slave nodes. When we measure the query processing time on a single server, there is no network cost because queries are requested and executed in the same server.

Fig. 13(b) shows the effect of different grouping techniques using the same extended vertex blocks (i.e., the guarantee of intra-partition processing is the same). The result indicates that the query processing depends on the replication level



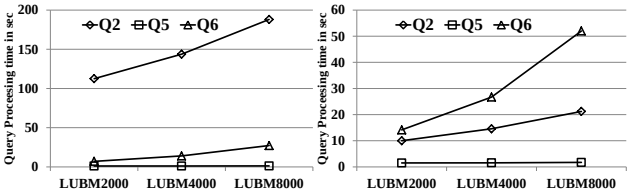
(a) LUBM2000 (b) SP2B-200M (c) DBLP (d) Freebase (e) DBpedia

Figure 10: Balance of generated partitions



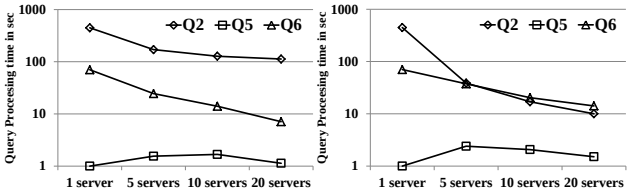
(a) LUBM2000 (b) SP2B-200M (c) DBLP (d) Freebase (e) DBpedia

Figure 11: Replication level



(a) 1-out-hash (b) 2-out-hash

Figure 14: Scalability with varying dataset size



(a) 1-out-hash (b) 2-out-hash

Figure 15: Scalability with varying server size

of the generated partitions. The query processing in the partitions generated using the minimum cut-based grouping technique is usually faster because the minimum cut-based technique generate smaller partitions than the others as shown in Fig. 11.

5.6 Scalability

To evaluate the scalability of our partitioning framework, we report the query processing results with varying dataset size in Fig. 14. For brevity, we choose one basic graph pattern query (Q6), one star-like query (Q5) and one complex query (Q2). The increase of the query processing time for Q6 is almost proportional to the dataset size and there is only slight increase for Q5 because its results are the same regardless of the dataset size. For Q2, there is only slight increase over 2-hop out-edge EVBs (Fig. 14(b)). However, there is a considerable increase over 1-hop out-edge EVBs because much more intermediate results are generated, compared to the increase of the final results.

Fig. 15 shows the results of another scalability experiment with varying numbers of slave nodes from 5 to 20 on LUBM2000. Note that the results on 1 server represent the query processing time without our partitioning and the query processing times are displayed as log scale. There is almost no big decrease for Q5 because it is already a fast query on 5 servers. We can see the considerable reduction of query processing time for Q2 and Q6 with an increasing number of servers, primarily due to the reduced partition size. However, as shown in the results of Q2 over 1-hop out-edge EVBs (Fig. 15(a)), there would be a point where adding more servers does not improve the query processing time any more because the transfer time of lots of results to the master node is unavoidable.

6. RELATED WORK

To process large graphs in a cluster of compute nodes, several graph computation models based on vertex centric approaches have been proposed in recent years, such as Pregel [17] and GraphLab [16]. Also, GraphChi [15] has been proposed to process large graphs on a single computer in reasonable time. Even though they can efficiently process some famous graph operations, such as page rank and shortest paths, they are not adequate for general graph pattern queries (i.e., subgraph matching) in which fast query processing (sometimes a couple of seconds) is preferred by evaluating small parts of input graphs. This is primarily because their approaches are based on multiple iterations and optimized for specific graph operations in which all (or most) vertices in a graph participate in the operations. Our partitioning framework focuses on efficient and effective partitioning for processing general graph pattern queries on large heterogeneous graphs.

Graph partitioning has been extensively studied in several communities for several decades [12, 9, 13, 14]. A typical graph partitioner divides a graph into smaller partitions that have minimum connections between them, as adopted by METIS [12, 14, 5] or Chaco[9, 3]. Various efforts in graph partitioning research have been dedicated to partitioning a

graph into similar sized partitions such that the workload of servers hosting these partitions will be more or less balanced. We utilize the results of one famous graph partitioner (METIS) to implement one of our grouping techniques to group our extended vertex blocks.

In recent years, a few techniques have been proposed to process RDF graphs in a cluster of compute nodes. [19, 11] directly store RDF triples (edges) in HDFS as flat text files to process RDF queries. [8] utilizes HBase, a column-oriented store modeled after Google's Bigtable, to store and query RDF graphs. Because general file system-based storage layers, such as HDFS, are not optimized for graph data, their query processing is much less efficient than those using a local graph processing engine, as reported in [10]. Also, because their query processing heavily depends on multiple rounds of inter-machine communication, they usually incur long query latencies. [10] utilizes the results of an existing graph partitioner to partition RDF graphs and stores generated partitions on RDF-3X to process RDF queries locally. As we reported in Sec. 5, running an existing graph partitioner has a large amount of overhead for huge graphs (or graphs having complex structure) and may not even be practically feasible for some large graphs.

7. CONCLUSION

We have presented VB-Partitioner – a distributed data partitioning model and algorithms for efficient processing of queries over large-scale graphs in the Cloud. This paper makes three original contributions. First, we introduce the concept of vertex blocks (VBs) and extended vertex blocks (EVBs) as the building blocks for semantic partitioning of large graphs. Second, we describe how VB-Partitioner utilizes vertex block grouping algorithms to place those vertex blocks that have high correlation in graph structure into the same partition. Third, we develop a VB-partition guided query partitioning model to speed up the parallel processing of graph pattern queries by reducing the amount of inter-partition query processing. We evaluate our VB-Partitioner through extensive experiments on several real-world graphs with millions of vertices and billions of edges. Our results show that VB-Partitioner significantly outperforms the popular random block-based data partitioner in terms of query latency and scalability over large-scale graphs.

Our research effort continues along several directions. The first prototype implementation of VB-Partitioner is on top of Hadoop Distributed File System (HDFS) with RDF-3X [18] installed on every node of the Hadoop cluster as the local storage system. We are interested in replacing RDF-3X by TripleBit [21] or GraphChi [15] as the local graph store to compare and understand how different choices of local stores may impact on the overall performance of our VB-Partitioner. In addition, we are working on efficient mechanisms for deploying and extending our VB-Partitioner to speed up the set of iterative graph algorithms, including shortest paths, PageRank and random walk-based graph clustering. For example, Pregel [17] can speed up the set of graph computations that are centered on out-vertex blocks such as shortest path discovery, and GraphChi [15] can speed up those iterative graph computations that rely on in-vertex blocks, such as PageRank and triangle counting. We conjecture that our VB-Partitioner can be effective for a broader range of iterative graph operations. Furthermore, we are also working on extending Hadoop MapReduce programming model and library to enable fast graph operations, ranging from graph queries, graph reasoning to iterative graph algorithms.

8. ACKNOWLEDGMENTS

This work is partially supported by grants from NSF Network Science and Engineering (NetSE) and NSF Trustworthy Cyberspace (SaTC), an IBM faculty award and a grant from Intel ISTC on Cloud Computing. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation and our industry sponsors.

9. REFERENCES

[1] About FacetedDBLP. *http://dblp.l3s.de/dblp++.php*.

[2] BTC 2012 Dataset. *http://km.aifb.kit.edu/projects/btc-2012/*.

[3] Chaco: Software for Partitioning Graphs. *http://www.sandia.gov/ bahendr/chaco.html*.

[4] DBpedia 3.8 Downloads. *http://wiki.dbpedia.org/Downloads38*.

[5] METIS. *http://www.cs.umn.edu/~metis*.

[6] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 120–124, New York, NY, USA, 2004. ACM.

[7] P. Barceló, L. Libkin, and J. L. Reutter. Querying graph patterns. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 199–210, New York, NY, USA, 2011. ACM.

[8] C. Franke, S. Morin, A. Chebotko, J. Abraham, and P. Brazier. Distributed Semantic Web Data Management in HBase and MySQL Cluster. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 105–112, Washington, DC, USA, 2011. IEEE Computer Society.

[9] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[10] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[11] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. on Knowl. and Data Eng.*, 23(9):1312–1327, Sept. 2011.

[12] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[13] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[14] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Supercomputing '98, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[15] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[18] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), Feb. 2010.

[19] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, DIDC '11, pages 35–44, New York, NY, USA, 2011. ACM.

[20] B. White and et al. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 255–270, 2002.

[21] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a Fast and Compact System for Large Scale RDF Data. *Proceedings of the VLDB Endowment*, 6(7), 2013.