Distributed Set Reachability

Sairam Gurajada
Max-Planck-Institute for Informatics
Campus E1.4
66123 Saarbrücken, Germany
gurajada@mpi-inf.mpg.de

Martin Theobald
University of Ulm
James-Franck-Ring O27
89069 Ulm, Germany
martin.theobald@uni-ulm.de

ABSTRACT

In this paper, we focus on the efficient and scalable processing of set-reachability queries over a distributed, directed data graph. A *set-reachability query* is a generalized form of a reachability query, in which we consider two sets S and T of source and target vertices, respectively, to be given as the query. The result of a set-reachability query are all pairs of source and target vertices (s, t), with $s \in S$ and $t \in T$, where s is reachable to t (denoted as $S \rightsquigarrow T$). In case the data graph is partitioned into multiple, edge- and vertex-disjoint subgraphs (e.g., when distributed across multiple compute nodes in a cluster), we refer to the resulting set-reachability problem as *distributed set reachability*. The key goal in processing a distributed set-reachability query over a partitioned data graph both efficiently and in a scalable manner is (1) to avoid redundant computations within the local compute nodes as much as possible, (2) to partially evaluate the local components of a set-reachability query $S \rightsquigarrow T$ among all compute nodes in parallel, and (3) to minimize both the size and number of messages exchanged among the compute nodes.

Distributed set reachability has a plethora of applications in graph analytics and for query processing. The current W3C recommendation for SPARQL 1.1, for example, introduces a notion of *labeled property paths* which resolves to processing a form of generalized graph-pattern queries with set-reachability predicates. Moreover, analyzing dependencies among *social-network communities* inherently involves reachability checks between large sets of source and target vertices. Our experiments confirm very significant performance gains of our approach in comparison to state-of-the-art graph engines such as Giraph++, and over a variety of graph collections with up to 1.4 billion edges.

1. INTRODUCTION

1.1 Background & Motivation

The reachability problem in directed graphs is one of the most fundamental problems in terms of both graph theory and applications: for a directed graph $G = (V, E)$ with vertices V and edges E, given a source vertex $s \in V$ and a target vertex $t \in V$, determine whether there is a consecutive path of edges from s to t over E.

To avoid redundant computations, many graph applications in fact require an extension of this basic reachability problem, where entire *sets S, T of source and target vertices*, respectively, need to be processed "at once". The resulting reachability problem, which we then coin *set reachability*, aims to retrieve all pairs of source and target vertices (s, t), with $s \in S$ and $t \in T$, where s is reachable to t. Moreover, in case the data graph is partitioned into multiple, edge- and vertex-disjoint subgraphs (e.g., when distributed across multiple compute nodes in a cluster), we refer to the resulting set-reachability problem as *distributed set reachability* (or "DSR" for short). The key goal in processing a DSR query over a partitioned data graph both efficiently and in a scalable manner is (1) to avoid redundant computations within the local compute nodes as much as possible, (2) to partially evaluate the local components of a set-reachability query $S \rightsquigarrow T$ among all compute nodes in parallel, and (3) to minimize both the size and number of messages exchanged among the compute nodes.

State-of-the-art indexing techniques for reachability queries [12, 16, 18, 25, 28, 32, 33, 34, 36] are largely limited to a centralized setting and thus address only point (1) of the above objectives. As for (2) and (3), we are currently aware of only one approach that specifically tackles the problem of distributed reachability queries for single-source, single-target queries [9]. For multi-source, multi-target (i.e., actual *set-*) reachability queries, we are aware of just two centralized approaches [12, 30] that provide suitable indexing and processing strategies. Both are based on a notion of *equivalence sets* among graph vertices which effectively resolves to a preprocessing and indexing step of the data graph to predetermine these sets. Efficient centralized approaches, however, are naturally limited to the main memory of a single machine and usually do not consider a parallel—in this case multi-threaded—execution of a reachability query.

Distributed graph engines, such as Google's Pregel [22], Berkeley's GraphX [35] (based on Spark [37]), Apache Giraph [1] and IBM's very recent Giraph++ [31], on the other hand, allow for the scalable processing of graph algorithms over massive, distributed data graphs. All of these provide generic API's for implementing various kinds of algorithms, including multi-source, multi-target reachability queries. However, a principal assumption we follow in this work is that set-reachability queries are *selective*. That is,

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915226

for any given sets S, T of source and target vertices, both S and T are usually much smaller than V, while the set of reachable pairs in turn usually is much smaller than the cross-product $S \times T$. Just like in relational approaches, an efficient processing of set-reachability queries thus calls for the aforementioned usage of indexing strategies that take advantage of the salient properties of the data graph. Graph indexing and the processing of selective queries however breaks the *node-centric computing* paradigm of Pregel and Giraph, where major amounts of the graph are successively shuffled through the network in each of the underlying MapReduce iterations (the so-called "supersteps").

Giraph++ is a very recent approach to overcome this overly myopic form of node-centric computing, which led to a new type of distributed graph processing that is coined *graph-centric computing* in [31]. By exposing intra-node state information as well as the inter-node partition structure to the local compute nodes, Giraph++ is a great step towards making these graph computations more context-aware. However, index structures that specifically tackle the iterative communication rounds required for the supersteps are difficult to accomplish even here, such that a direct implementation of a reachability query may still result in as many iterations (and hence communication rounds) as the diameter of the graph in the worst case.

Finally, the set-reachability problem has a plethora of applications in graph analytics and query-processing tasks. With its recent update, SPARQL 1.1, for example, underwent a major revision in which the usage of *labeled property paths* [2] allows a user to formulate transitive reachability constraints among the query variables. Since both the source and target variables of a property path may become bound to multiple RDF constants at query processing time, the processing of property paths in SPARQL 1.1 resolves to processing set-reachability queries. Another interesting application of set-reachability is *community analysis in social networks*. That is, given sets of source and target vertices, each representing social-net users such as on Twitter or Facebook, we may want to efficiently detect which communities are densely connected. For example, consider two communities—billionaires and non-profit organizations—, it would be interesting to find the list of billionaires who are also involved in philanthropic activities.

1.2 Contributions

We summarize the contributions of our work as follows.

- We formalize the problem of *distributed set reachability* (DSR) over a partitioned, and hence distributed, directed data graph. To our knowledge, our approach is the first to specifically tackle this problem.
- We develop a graph-based index structure that allows us to strictly restrict the communication protocol among the compute nodes to a *single round of message exchange* in order to resolve the results of any DSR query posed against a given partitioning of the data graph. This guarantee holds regardless of the properties of the data graph (such as its diameter and partition structure) and the properties of the query (such as the distribution of source and target vertices among the graph partitions).
- Our indexing strategy allows for *incremental updates* of the underlying data graph, with an efficient support for vertex and edge insertions and a principle support for respective deletions.
- Our approach is also *extensible* in the sense that any existing, centralized reachability index can be "plugged-in" at the local compute nodes. We report the results of our distributed approach in combination with a plain DFS search [6], the MS-BFS approach of [30], and FERRARI [28] as local search strategies.
- Moreover, we provide an *extensive experimental evaluation* of our approach over a variety of both small and large graphs and in comparison to different extensions of Giraph++. We also investigate two application scenarios of our approach for processing SPARQL 1.1 queries with property paths and for detecting dependencies among social-network communities.

2. PRELIMINARIES

We start by formally defining our data and query model. This section also serves to establish the notation we will use through the rest of the paper.

DEFINITION 1. *A* **data graph** $G(V, E, L, \phi)$ *is a directed graph consisting of vertices* V *and edges* $E \subseteq V \times V$*. Further, we assume a unique label (i.e., an identifier) for each vertex in the graph to be given by a bijective mapping* $\phi : V \to L$ *from vertices* V *to labels* L*.*

Given two vertices $s, t \in V$, a *path* P from s to t is denoted by a consecutive set of edges $\{(s := u_0, u_1), (u_1, u_2), \ldots, (u_{n-1}, u_n =: t)\}$ such that each $(u_i, u_j) \in E$. Two vertices s and t are called *reachable*, denoted as $s \rightsquigarrow t$, iff there exists at least one path $P \subseteq E$ from s to t.

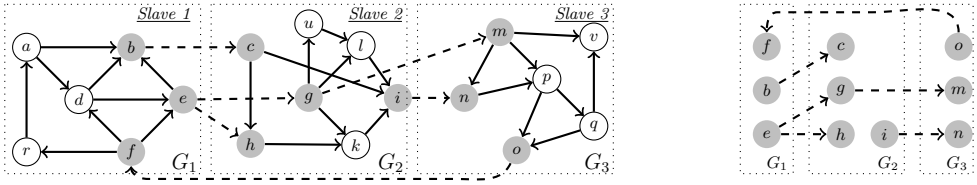
Graph Partitioning. To scale a reachability query $s \rightsquigarrow t$ to a very large data graph, which may not necessarily fit into the main-memory of a single compute node, we allow the graph to be partitioned into k edge- and vertex-disjoint subgraphs. Formally, a *subgraph* $G_i(V_i, E_i, L, \phi)$ of a data graph $G(V, E, L, \phi)$ is a vertex-induced subgraph of G, such that $V_i \subseteq V$ and $E_i = \{(u, v) \,|\, u \in V_i, v \in V_i \text{ and } (u, v) \in E\}$. We refer to $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ as the *partitioning* of G and to $C(V_C, E_C, L, \phi)$ as the *cut* of G, respectively. Here, k denotes the number of partitions of G and each partition G_i (including C) denotes a subgraph of G. Moreover, C is a subgraph of G such that, for a given graph partitioning $\mathcal{G}$, $E_C = \{(u, v) \,|\, (u, v) \in E, u \in V_i, v \in V_j \text{ and } i \neq j\}$ with vertices $u, v \in V_C$, iff edge $(u, v) \in E_C$.

DEFINITION 2. *Given a partitioning* $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$ *and the respective cut* $C(V_C, E_C, L, \phi)$ *of a data graph* G*, a* **DSR query***, denoted as* $S \rightsquigarrow T$*, returns all pairs of source and target vertices* (s, t)*, with* $s \in S$ *and* $t \in T$*, where* $s \rightsquigarrow t$*.*

DEFINITION 3. *For a given graph partitioning* $\mathcal{G}$ *and implied cut* C *of a data graph* G*, we define the set of* **in-boundaries** I_i *for partition* G_i *as* $I_i = \{v \,|\, v \in V_i, \exists (u, v) \in E_C, u \in V_j \text{ and } i \neq j\}$*, i.e., as the set of vertices in* G_i *that have an incoming edge from the cut* C*.*

Conversely, we define the set of **out-boundaries** $O_i = \{v \,|\, v \in V_i, \exists (v, u) \in E_C, u \in V_j \text{ and } i \neq j\}$ *as the set of vertices in* G_i *that have an outgoing edge into the cut* C*.*

Partitioning Function. We denote by $\rho : V \mapsto \mathbb{N}_0^+$ the *partitioning function* that determines to which of the compute nodes (i.e., "slaves") in a cluster architecture each graph vertex $v \in V$ is distributed. Without loss of generality, and to simplify notation for the following presentation, we assume a simple partitioning strategy by distributing every



(a) Graph G (b) Cut C

Figure 1: (a) Graph G with partitions $\mathcal{G} = \{G_1, G_2, G_3\}$ and (b) respective cut C

vertex $v \in V_i$ to slave i (i.e., $\rho(v) = i$ for each $v \in V_i$). We will thus refer to a graph partition and a slave interchangeably. To increase concurrent executions (e.g., when using multi-threading at the local compute nodes), an "overpartitioning" strategy may be employed instead, by assigning multiple graph partitions to each of the slaves.

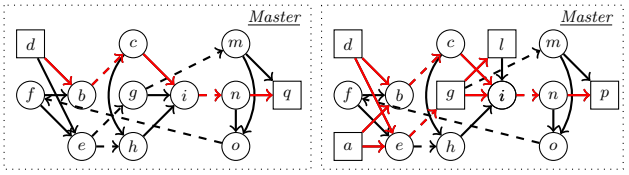
EXAMPLE 1. *Figure 1(a) shows an example graph G with three partitions $\mathcal{G} = \{G_1, G_2, G_3\}$ which are stored at three slaves. Its corresponding cut C is shown in Figure 1(b). In- and out-boundaries are $I_1 = \{f\}, O_1 = \{b, e\}$, $I_2 = \{c, g, h\}$, $O_2 = \{i\}$, and $I_3 = \{m, n\}$, $O_3 = \{o\}$, respectively.*

Distributed Reachability. Our approach for processing DSR queries uses a similar setting as described by Fan et al. [9]. In [9], a distributed (single-source, single-target) reachability query $s \rightsquigarrow t$ over a master-slave architecture is evaluated as follows. The master receives $s \rightsquigarrow t$ and communicates it to all slaves. At partition G_i, containing the source s, a local evaluation of the reachability of s to each vertex in the set of out-boundaries O_i is computed first. Similarly, at partition G_j containing the target t, a local evaluation of the reachability of each vertex in the set of in-boundaries I_j is computed next. Additionally, a local computation of the reachability between all in-boundaries I_i and out-boundaries O_i is computed at each partition $G_1, \ldots, G_k$ and hence at all slaves $i = 1..k$ in the compute cluster *in parallel*.[1]

The resulting local reachability information is encoded into sets of Boolean formulas, where each such set represents the local connectivities between the in-boundaries I_i (including the source s, if present) with the out-boundaries O_i (including the target t, if present) at partition G_i. All of these formulas are communicated back to a single master node for the final evaluation. A query-specific *global dependency graph* is constructed at this master node for $s \rightsquigarrow t$ using the Boolean formulas and the static cut C. A reachability algorithm is then run over the dependency graph to answer $s \rightsquigarrow t$ via substitution of the variables that are reversely connected to the target vertex t. The overall algorithm provided in [9] can be implemented with a simple communication protocol and, for example, be executed in a single MapReduce iteration among all compute nodes.

EXAMPLE 2. *Consider the distributed reachability query $d \rightsquigarrow q$ over the graph partitioning shown in Figure 1. The local evaluation at each partition results in the following Boolean representation of partial reachability information: $\{d = b \vee e, f = b \vee e\}@G_1$, $\{c = i, g = i, h = i\}@G_2$, $\{m = q \vee o, n = q \vee o\}@G_3$. By including the edges in the cut C (Figure 1(b)), the global dependency graph (Figure 2(a)) is constructed at the master node to finally resolve $d \rightsquigarrow q$. By running a reachability algorithm (such as backward DFS) over the dependency graph, one can find that $d \rightsquigarrow q$ is indeed* true *(the red path in Figure 2(a)).*

[1]Note that we follow a slightly different definition of in- and out-boundaries than in [9]. However, the algorithm in [9] directly translates to the one outlined above.



(a) Single Reachability (b) Set Reachability

Figure 2: Dependency graph as constructed in [9] for a single reachability query (a) and an extension to set reachability (b)

3. DISTRIBUTED SET REACHABILITY

3.1 Naïve Approach

A *naïve approach* to extend the distributed reachability problem to sets of vertices S, T would be to simply invoke a separate reachability query $s \rightsquigarrow t$ for every pair (s, t), with $s \in S$ and $t \in T$. However, an obvious reason for the limited efficiency of this approach, even for reasonably-sized sets S and T, is that this approach does not exploit the assumption we made earlier, namely that queries are *selective*, i.e., that by far not all pairs in $S \times T$ are reachable. Consequently, this approach can also not reuse any intermediate computations and thus likely performs many redundant computations.

3.2 DSR with a Dynamic Dependency Graph

An *improved approach* to extend the distributed reachability algorithm provided in [9] to sets is as follows. Let $S \rightsquigarrow T$ be the query received at the master. First, we partition $S \rightsquigarrow T$ into subqueries $S_1 \rightsquigarrow T_1$, $S_2 \rightsquigarrow T_2$, $\ldots$, $S_k \rightsquigarrow T_k$, where k is the number of graph partitions, such that each $S_i \subseteq V_i$ and $T_i \subseteq V_i$ contains only vertices that are local to partition G_i. Next, a local evaluation at each slave i involves finding the reachability among all pairs of vertices from the sets $S_i \cup I_i$ and $O_i \cup T_i$, respectively. These can again be run *in parallel* across all slaves. The resulting reachability information, again represented as sets of Boolean formulas, is then communicated from all slaves to the master node for the final evaluation. At the master node, the query-specific dependency graph for the sets S, T is constructed as described in Section 2, and a local reachability algorithm is then used at the master node to emit all reachable pairs (s, t), with $s \in S$ and $t \in T$.

EXAMPLE 3. *Consider the DSR query $S \rightsquigarrow T$ with $S = \{a, d, g\}$ and $T = \{l, p\}$ over the cut C shown in Figure 1(b). The sets of Boolean formulas obtained after the local evaluation at each slave are as follows: $\{a = b \vee e, d = b \vee e, f = b \vee e\}@G_1$, $\{c = i, g = i \vee l, h = i\}@G_2$, $\{m = p \vee o, n = p \vee o\}@G_3$. At the master node, after evaluating $S \rightsquigarrow T$ over the global dependency graph shown in Figure 2(b), we obtain the following reachable pairs of source and target vertices: $\{(a, l), (a, p), (d, l), (d, p), (g, l), (g, p)\}$.*

3.2.1 Discussion

Although the second algorithm provides a more viable solution of the DSR problem than the naïve approach, it still leaves a number of disadvantages that limit both its efficiency and scalability.

- First, the query-dependent, global dependency graph is generated "from scratch" for each query $S \rightsquigarrow T$, although both the cut C and the local reachability information $I_i \rightsquigarrow O_i$ among the in- and out-boundaries at each graph partition G_i are in fact static.
- Second, the approach does not leverage any distributed computation in its second step, as the final reachability computation $S \rightsquigarrow T$ over the global dependency graph is performed only by a single master node.
- Third, since the global dependency graph is generated dynamically for each query $S \rightsquigarrow T$, a reachability index for the static cut C and the local $I_i \rightsquigarrow O_i$ components cannot be constructed, which restricts the final reachability computation to either a simple BFS or DFS strategy over the dependency graph.

3.3 DSR with a Static Reachability Index

In *our approach*, instead of computing the global dependency graph for each incoming query from scratch at the master node, we precompute a partition-specific variant thereof, called the "boundary graph", only once and store this in the form of a static reachability index at each slave. This strategy provides multiple benefits. First, it avoids repeated computations of the boundary graph for each query. Second, since each slave has the complete reachability information among the boundary vertices of all other slaves available, finding the reachability of any two vertices (s, t) in the entire data graph G resolves to a local reachability computation at at most two slaves, which is irrespective of the diameter of the graph and the distribution of the source and target vertices of a set-reachability query (see Theorems 1 and 2). Additionally, an index can be built over the static boundary graph to accelerate this processing. Third, storing a (compacted version of the) boundary graph at each slave allows for a fully distributed processing of a set-reachability query and thus avoids the single-node bottleneck of previous approaches. We next formally define how we generate the boundary graph and its derived index structures.

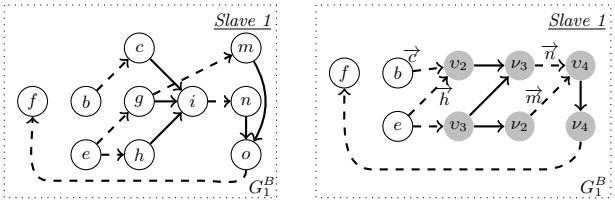
3.3.1 Precomputed Index Structures

Boundary Graph. A *boundary graph* is a directed graph that represents the reachability information among the in- and out-boundaries of all graph partitions $\mathcal{G} = \{G_1, \ldots, G_k\}$ with respect to a given cut C.

Definition 4. *Let $G_i^B(V_i^B, E_i^B, L, \phi)$ denote the* ***boundary graph*** *we compute for partition G_i, such that the following holds:*

- *The vertices $V_i^B = \bigcup_{i=1..k} I_i \cup O_i$ consist of the union of all in- and out-boundaries of all partitions $G_1, \ldots, G_k$.*
- *There exists an edge $(u, v) \in E_i^B$, iff*
  - *$(u, v) \in E_C$, or*
  - *$u \in I_j$ and $v \in O_j$, for $j \neq i$, and $u \rightsquigarrow v$ (i.e., u and v are both located at another partition G_j and there exists a path from u to v in G_j).*

That is, the boundary graph for partition G_i merges the static cut C with the static reachability information $I_j \rightsquigarrow O_j$ among all the remaining graph partitions G_j (for $i \neq j$) into a new, precomputed graph G_i^B. The resulting boundary graphs are thus partition-specific.



(a) Non-optimized (b) Optimized

Figure 3: Boundary graph G_1^B for partition G_1

Example 4. *For our graph G with partitions G_1, G_2, G_3 and respective cut C as shown in Figure 1, the boundary graph G_1^B for partition G_1 is shown in Figure 3(a). Here, the dashed edges refer to edges in the cut C, while the solid edges denote the transitive reachability $I_j \rightsquigarrow O_j$ (for $j \neq 1$).*

Complexity. The construction of the boundary graph requires us to materialize the pairwise reachability $I_i \rightsquigarrow O_i$ among the in- and out-boundaries for each partition G_i. Using a simple BFS/DFS-based approach, the time complexity of this computation is $\mathcal{O}((|V_i|+|E_i|)\cdot|I_i|\cdot|O_i|)$ per partition. This can be further improved to $\mathcal{O}(1 \cdot |I_i| \cdot |O_i|)$ when using a sophisticated, local reachability index for this operation. On the other hand, the (worst-case) space complexity for storing the boundary graph at partition i is $\mathcal{O}(\sum_{j=1}^{k} |I_j| \cdot |O_j| + |E_C|)$, for $j \neq i$. From this, one can deduce that both the time and space complexity of the boundary graph computation strongly depend on the amounts of in- and out-boundaries we obtain from the cut C.

Min-k-Cut Partitioning. A standard approach to reduce this number of boundary vertices is to reduce the number of edges in the cut C, while trying to keep the sizes of the partitions $G_1, \ldots, G_k$ balanced. Although finding an optimal such *min-k-cut* partitioning is a well-known NP-complete problem [6], current graph libraries such as METIS [17] are capable of achieving very good approximations even for graphs with hundreds of millions of edges.

Equivalence Sets. Even for a given cut C, we can further reduce the size of the boundary graph by grouping the in- and out-boundary vertices into *equivalence sets* of vertices, thus continuing the idea presented in [12] to a distributed setting. Specifically, we achieve this by grouping the boundary vertices into *forward-* and *backward-equivalent sets* according to the following definition.

Definition 5. *Two in-boundaries b_1, b_2 are called* ***forward-equivalent*** *with respect to subgraph G_i, i.e., $b_1 \equiv^f b_2$, iff for any vertex $v \in V_i - I_i$ and $b_1 \rightsquigarrow v$, it holds that $b_2 \rightsquigarrow v$.*

Conversely, two out-boundaries b_1, b_2 are called ***backward-equivalent*** *with respect to subgraph G_i, i.e., $b_1 \equiv^b b_2$, iff for any vertex $v \in V_i - O_i$ and $v \rightsquigarrow b_1$, it holds that $v \rightsquigarrow b_2$.*

That is, once the forward- and backward-equivalent sets of vertices are identified for each subgraph G_i, each such set is replaced by a new *in-virtual vertex* υ (for a forward-equivalent set) and a new *out-virtual vertex* ν (for a backward-equivalent set), respectively.

Example 5. *Following the above definition of equivalence for the partitioning $\mathcal{G} = \{G_1, \ldots, G_3\}$ of G shown in Figure 1(a), we can obtain the following, partition-specific equivalence sets: $\{\upsilon_1 = \{f\}\}@G_1$, $\{\nu_1 = \{b, e\}\}@G_1$; $\{\upsilon_2 =$*

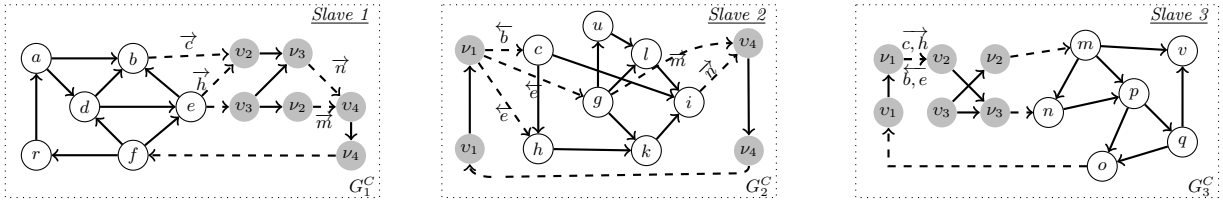


Figure 4: Final compound graphs G_1^C, G_2^C, G_3^C constructed for graph G with cut C of Figure 1

$\{c,h\}, \upsilon_3 = \{g\}\}@G_2$, $\{\nu_2 = \{g\}, \nu_3 = \{i\}\}@G_2$; $\{\upsilon_4 = \{m,n\}\}@G_3$, $\{\nu_4 = \{o\}\}@G_3$.

Next, the in- and out-boundaries are redefined with respect to the new virtual vertices. That is, I_i comprises of all in-virtual vertices and O_i comprises of all out-virtual vertices. The optimized boundary graph for partition G_1 is shown in Figure 3(b). Note that we attach additional labels to the cross-edges in the boundary graph to obtain a lossless representation of the boundary graph with respect to the partitions $G_1, \ldots, G_k$. For example, the cross-edge (b, c) is represented by connecting the vertex b and the in-virtual vertex υ_2 with the label $\overrightarrow{c}$ to denote that b is connected to only c in υ_2. The forward arrow denotes that this connection is valid only for a forward exploration. This is required, since vertices c, h are forward-equivalent, i.e., $c \equiv^f h$, with respect to partition G_2 only.

Computing Equivalence Sets. According to Definition 5, two in-boundaries $b_1 \in I_i$ and $b_2 \in I_i$ are forward-equivalent if they are reachable to exactly the same set of vertices in $V_i - I_i$. To determine the sets of forward-equivalent boundaries, we need to (1) compute all reachable pairs from I_i to $V_i - I_i$ and then (2) group the vertices in I_i into these equivalence sets. For large sets I_i and $V_i - I_i$, this computation may be prohibitively expensive. To address (1) and thus reduce the input that needs to be considered for (2), we apply the following optimizations.

- b_1, b_2 can only be forward-equivalent with respect to partition G_i if both belong to the same *strongly connected component* (SCC) in G_i. We thus condense G_i into a more compact DAG by computing the SCCs over G_i.
- Instead of considering all target vertices $V_i - I_i$, we consider only the *direct successors* $S(I_i)$ of I_i, and hence $S(I_i) - I_i$, to check for forward-equivalence. The intuition for considering only successors is that if two boundaries b_1, b_2 are reachable to the same set of vertices in $S(I_i) - I_i$, then b_1, b_2 also are reachable to the same set of vertices in $V_i - I_i$.

A similar construction then holds also for backward-equivalence, except that *predecessors* $P(O_i)$ are considered instead.

Example 6. *Consider partition G_3 with in-boundary set $I_3 = \{m, n\}$ to compute the sets of forward-equivalent vertices in I_3. In this case, this requires us to only verify whether $m \equiv^f n$, since m, n are the only in-boundaries in I_3. First, we run the SCC algorithm to condense G_3 into the DAG G_3'. In this example, $G_3' = G_3$, and we see that m, n do not belong to the same SCC. We then check their forward-equivalence based on the sets of vertices in $V_3 - I_3$ that are reachable from both m and n. To compute these reachable sets of vertices, we consider only the direct successors $S(I_3) - I_3 = \{p, v\}$ instead of considering all of $V_3 - I_3 = \{p, o, q, v\}$. Thus, the reachable set of vertices of both m and n is $\{p, v\}$, and hence we have $m \equiv^f n$.*

A compact algorithm for computing these equivalence sets is depicted in Appendix 8.1.

Compound Graph. After compacting the partition-specific boundary graphs G_i^B by replacing both the forward- and backward-equivalent sets of vertices with their in- and out-virtual counterparts, we perform one more step to obtain our final graph index for evaluating DSR queries. To do so, we merge the partition-specific boundary graphs with the local partitions into a *compound graph* G_i^C for each partition G_i. These compound graphs will facilitate the processing of DSR queries via a combination of local reachability computations and a single filtering step among these local results.

Definition 6. *Let $G_i^C(V_i^C, E_i^C, L, \phi)$ denote the **compound graph** we compute for partition G_i, such that the following holds:*

- *The vertices $V_i^C = V_i \cup V_i^B$ consist of the union of vertices in the local subgraph G_i and boundary graph G_i^B.*
- *The edges $E_i^B = E_i \cup E_i^B$ consist of the union of edges in the local subgraph G_i and boundary graph G_i^B.*

Figure 4 shows the compound graphs for the initial data graph G from Figure 1(a).

Forward- and Backward-Lists. Our last precomputation step consists of storing the forward- and backward-lists, F_i and B_i, of boundaries which are non-local to each partition G_i. These will serve for routing messages to only those partitions G_j which are connected to G_i. Specifically, the *forward-list* $F_i = \bigcup_{j \neq i} \{\upsilon \mid \upsilon$ *is in-virtual vertex of* $G_j\}$ is the set of all vertices that are non-local to G_i and are in-virtual vertices of another partition G_j. Similarly, the *backward-list* $B_i = \bigcup_{j \neq i} \{\nu \mid \nu$ *is out-virtual vertex of* $G_j\}$ consists of all out-virtual vertices that are non-local to G_i. For instance, for partition G_1 shown in Figure 4, we have $F_1 = \{\upsilon_2, \upsilon_3, \upsilon_4\}$ and $B_1 = \{\nu_2, \nu_3, \nu_4\}$.

3.3.2 Evaluating DSR Queries

Given these precomputed index structures, i.e., the compound graphs G_i^C and respective forward- and backward-lists, F_i and B_i, evaluating a DSR query now becomes straightforward. We again begin with a discussion of the single-source, single-target case and then explain how it generalizes to the multi-source, multi-target case.

A. Single Reachability. Consider the reachability query $s \rightsquigarrow t$. The algorithm for processing the query is shown in Algorithm 1. Given a data graph G with partitioning $\mathcal{G}$, we evaluate the query as follows. If both s and t belong to same partition G_i, then the reachability $s \rightsquigarrow t$ is confined to only slave i which stores the compound graph G_i^C. Since the compound graph G_i^C augments each G_i with the global reachability information among all boundary vertices, we can safely evaluate the reachability of $s \rightsquigarrow t$ on G_i^C by calling any centralized reachability algorithm via the function localSetReachability(.) (Lines 11-13). A formal justification for this is provided by the following theorem.

Theorem 1. *Let s, t both be local vertices of partition i, i.e., $s, t \in V_i$. Then the evaluation of the reachability*

Algorithm 1: Distributed Reachability Processing

Input: Compound graphs: $\{G_1^C, G_2^C, ..., G_k^C\}$, Query: $s \rightsquigarrow t$
Output: true/false

```
1  Master:
2  rank_s := ρ(s)                    ▷ i.e., s ∈ V_i and G_i is at Slave ρ(i)
3  rank_t := ρ(t)                    ▷ i.e., t ∈ V_j and G_j is at Slave ρ(j)
4  result := false
5  foreach rank do
6      result := result ∨ compute(s, rank_s, t, rank_t)
       ▷ invokes parallel computations at all ranks
7  return result

8  Slave i:
9  method compute(s, rank_s, t, rank_t) :
10 rset := ∅
11 if rank_s = i and rank_t = i then
       ▷ invoke local reachability evaluation
12     if localSetReachability({s}, {t}) ≠ ∅ then
13         return true
14 else if i = rank_s then
15     j := rank_t
16     Υ_s^j := localSetReachability({s}, F_i^j);
       ▷ F_i^j ⊆ F_i is the set of in-virtual vertices local to j
17     rset[s] := Υ_s^j
18     sendMessage(j, rset)
19     return false
20 else if i = rank_t then
21     receiveMessage(i, rset)
22     Υ_s^i := rset[s]
23     for v in Υ_s^i do
24         b := v.rep                ▷ b is a member vertex in eqset v
25         if localSetReachability({b}, {t}) ≠ ∅ then
26             return true
27 return false
```

$s \rightsquigarrow t$ over graph G can be answered entirely locally over the compound graph G_i^C without requiring any message exchange among the compute nodes.

Proof. See Appendix 8.2.1. □

Example 7. *Consider the query $b \rightsquigarrow f$. Both vertices b, f are local to partition G_1. By considering only the subgraph G_1, one cannot find that f is reachable from b. But by considering the whole graph G, we see that $b \rightsquigarrow f$ is* true *via the path $b \to c \to i \to n \to p \to o \to f$. However, using the local compound graph G_1^C (see Figure 4), we can indeed find that $b \rightsquigarrow f$ is* true *via the path $b \to \upsilon_2 \to \nu_3 \to \upsilon_4 \to \nu_4 \to f$.*

If, on the other hand, s and t are located at two different partitions G_i, G_j, with $i \neq j$, the evaluation of a reachability query works as follows (Lines 14-25). Starting at partition G_i, we find the reachability from s to all the forward-boundaries $\upsilon \in F_i^j \subseteq F_i$ (Line 15) which are located at another partition G_j. Let $\Upsilon_s^j \subseteq F_i$ be the set of in-virtual vertices located at partition G_j (and hence stored by slave j as per our assumption) which are reachable from s. The message rset[s]:=$\langle s, \Upsilon_s^j \rangle$ is then communicated to slave j. At slave j, we consider each $\upsilon \in \Upsilon_s^j$ and replace it with any one of its members b, after which we evaluate the reachability from b to the local target vertex t. If there exists one such $b \in \upsilon \in \Upsilon_s^j$ with $b \rightsquigarrow t$, we report that $s \rightsquigarrow t$ is *true* (Lines 22-25).

Theorem 2. *Let $s \in V_i$ and $t \in V_j$, with $i \neq j$. Then, the evaluation of the reachability $s \rightsquigarrow t$ can be answered over the two compound graphs G_i^C and G_j^C by using a single step of message exchange from slave i to slave j.*

Proof. See Appendix 8.2.2 □

Example 8. *Consider the query $a \rightsquigarrow q$, where a is located at partition G_1 and q is located at partition G_3. At partition G_1, we compute the reachability from a to the single forward-boundary $\{\upsilon_4\}$ which is located at G_3. From the compound graph G_1^C (shown in Figure 4), we have $\Upsilon_a^3 = \{\upsilon_4\}$ since $a \rightsquigarrow \upsilon_4$. Υ_a^3 is then communicated to slave* 3*. At slave* 3*, we expand the actual vertices represented by the virtual vertex υ_4 (say m, since $m \in \upsilon_4$) and find the reachability from m to q. Since $m \rightsquigarrow q$, we thus find that $a \rightsquigarrow q$ is* true*.*

B. Set Reachability. An actual DSR query $S \rightsquigarrow T$, which is received by the master node, is processed in our approach as shown in Algorithm 2. First, $S \rightsquigarrow T$ is partitioned into subqueries $S_1 \rightsquigarrow T_1$, $S_2 \rightsquigarrow T_2$,..., $S_k \rightsquigarrow T_k$, where k is again the number of graph partitions. The partitioning of the query into these subqueries is determined such that each source vertex $s_i \in S_i$ and target vertex $t_i \in T_i$ resides locally at partition G_i (Line 2).

Step 1. (Lines 13-19) A local evaluation at partition G_i involves processing the pairwise reachability among the vertices from S_i to T_i and from S_i to F_i at *all slaves $i = 1..k$ in parallel*. This operation generates two types of reachable pairs: (s_i, t_i) and (s_i, υ_j). The first type denotes the reachability between both a local source $s_i \in S_i$ and a local target $t_i \in T_i$. The second type denotes the reachability between a local source $s_i \in S_i$ and a forward-boundary $\upsilon_j \in F_i$, which is represented by an in-virtual vertex located at slave j.

Step 2. (Lines 21-32) The communication of the remotely reachable pairs, each of the form (s_i, υ_j), is performed from slave i to slave j among *all pairs of slaves $i, j = 1..k$ in parallel*. In order to reduce the overhead of communicating individual pairs, each slave buffers its partial reachability information and communicates this buffer at once. Each buffer sent from slave i to slave j is of the form $\{\langle s_i, \Upsilon_{s_i}^j \rangle\}$ for all $s_i \in S_i$. For easier processing, the messages received at slave i from all other slaves are stored in an inverted index $\mathcal{I}_i(\Upsilon_*^i, L_i)$, where Υ_*^i is the aggregated set of in-virtual vertices (local to slave i). For each in-virtual vertex $\upsilon \in \Upsilon_*^i$, its aggregated non-local source set $S_\upsilon \subseteq S$ is stored in L_i. That is, for $s \in S_\upsilon$ and $\upsilon \in \Upsilon_*^i$, we already know that $s \rightsquigarrow \upsilon$.

Step 3. (Lines 34-39) A final local evaluation involves processing the set reachability $\Upsilon_*^i \rightsquigarrow T_i$ from the in-virtual vertices Υ_*^i to the target sets T_i at *all slaves $i = 1..k$ in parallel*. For each in-virtual vertex $\upsilon \in \Upsilon_*^i$ and original vertex b represented by υ, we evaluate the reachability from b to all targets $t \in T_i$. If $b \rightsquigarrow t$ is *true*, then for each $s \in S_\upsilon$, we report that $s \rightsquigarrow t$ is *true*.

Example 9. *Consider again the graph G with partitions G_1, G_2, G_3 in Figure 1(a). The respective compound graphs G_1^C, G_2^C, G_3^C are shown in Figure 4. Let $S = \{d, l, p\} \rightsquigarrow T = \{a, k, q\}$ be the DSR query received at the master node. The query is partitioned into $\{d\} \rightsquigarrow \{a\}$, $\{l\} \rightsquigarrow \{k\}$, $\{p\} \rightsquigarrow \{q\}$. At partition G_1, we find the set-reachability (Step 1) between $\{d\}$, $\{\upsilon_2, \upsilon_3, \upsilon_4, a\}$, thus returning the reachable pairs $\{(d, \upsilon_2)$, (d, υ_3), (d, υ_4), $(d, a)\}$. We perform the same operation in parallel at slaves* 2 *and* 3 *and communicate the results to all other slaves (Step 2). At slave* 1*, we receive*

Algorithm 2: Distributed Set-Reachability Processing

Input: Compound graphs $\{G_1^C, G_2^C, ..., G_k^C\}$, Query: $S \rightsquigarrow T$
Output: R $\{(s,t) \mid s \in S, t \in T \text{ and } s \rightsquigarrow t\}$

```
1  Master:
2  partition S, T into {(S_1,T_2),(S_2,T_2),...,(S_k,T_k)}
     ▷ where S_i ⊆ V_i and T_i ⊆ V_i
3  result := ∅
4  for i = 1...k do
5      result := result ∪ compute(S_i, T_i)
6  return result

7  Slave i:
8  method compute(S_i, T_i) :
9  local_rset := ∅
10 remote_rset := ∅
11 result := ∅
12 // Step 1:
13 local_rset := localSetReachability(S_i, T_i)
14 remote_rset := localSetReachability(S_i, F_i)
15 for s in S_i do
16     for t in local_rset[s] do
17         result := result ∪ {(s,t)}
18     for v in remote_rset[s] do
19         Υ_s^j := Υ_s^j ∪ v
           ▷ v is an in-virtual vertex of partition j
20 // Step 2:
21 for j = 1 to k do
22     if j ≠ i then
23         msg := ∅
24         for s in S_i do
25             msg := msg ∪ {⟨s, Υ_s^j⟩}
26         sendMessage(j, msg)
27 I_i(Υ_*^i, L_i) = ∅
28 for j = 1 to k do
29     receiveMessage(j, msg)
30     for ⟨s, Υ_s^i⟩ in msg do
31         for v in Υ_s^i do
32             I_i[v] := I_i[v] ∪ {s}
33 // Step 3:
34 for v in Υ_*^i do
35     b := v.rep
36     local_rset := localSetReachability({b}, T_i)
37     for s in I_i[v] do
38         for t ∈ local_rset[b] do
39             result := result ∪ {(s,t)}
40 return result
```

the following reachability information: $\{(v_1, [l,p])\}$*. Similarly, at slave* 2*, we receive* $\{(v_2, [d,p]), (v_3, [d,p])\}$*; and at slave* 3*, we receive* $\{(v_4, [d,l])\}$*. At the end of the local evaluation from boundaries to the final targets (Step 3), by replacing virtual vertices with each of their represented vertices (at slave* 1*,* v_1 *is replaced with* f*), the sets* $\{(d,a),(l,a),(p,a)\}@G_1$*,* $\{(d,k),(l,k),(p,k)\}@G_2$ *and* $\{(d,q),(l,q),(p,q)\}@G_3$ *of reachable pairs are generated at the partitions.*

Local Reachability Evaluation. Algorithms 1 and 2 both require partial reachability processing at each slave via the function localSetReachability(.). For this, any centralized reachability index (see, e.g., [6, 28, 12, 36]) can be plugged into our framework. We abstract this by calling localSetReachability(.) in our algorithms whenever a local (set-)reachability operation is invoked.

Forward vs. Backward Processing. Our above discussion focused on starting from the source vertices and ending at the target vertices. If there are less targets than sources, one may also start from the target vertices and search backwards to the source vertices to arrive at the same results. We therefore maintain both forward- and backward-lists, F_i and B_i, to facilitate these two directions of searching.

3.3.3 Incremental Updates

Insertions. Insertions over the SCC-condensed compound graphs G_i^C can be implemented *without* storing the original (i.e., uncondensed) compound graphs.

Let (u,v) denote a new edge that is to be inserted into the graph G. First, assume both u and v belong to the same graph partition i. Further, if u, v belong to the same SCC, then adding (u,v) to G_i would not change the local compound graph G_i^C (nor any other) at all and thus can be safely ignored. If, on the other hand, u, v belong to two different SCCs, then a series of update actions are required. First, we add the new edge to the local compound graph G_i^C and locally recompute the SCCs and equivalence sets. Next, new connections among the local in- and out-boundaries, I_i and O_i, are communicated to all other partitions j (for $j \neq i$) as additional edges. These can be incrementally merged into all the compound graphs G_j^C by updating their SCCs as well. Second, if u and v belong to two different partitions i and j, then this means we have a new edge in the cut C, which however does not affect the reachability within partitions i and j. Thus, (u,v) can directly be merged into the distributed compound graphs as described above.

Let n, m denote the number of vertices and edges in the condensed compound graph G_i^C, and let $|I_i|$, $|O_i|$ be the number of in- and out-boundaries for partition i, respectively. By adding a local edge to partition i, a partial or full recomputation of the connections among vertices from I_i to O_i is required. Thus the worst-case time complexity of this step is $\mathcal{O}((n+m) \cdot |I_i| \cdot |O_i|)$, which is asymptotically optimal [7]. The SCC recomputation at each compound graph has a time complexity of $\mathcal{O}(n' + m')$, where n' and m' are the numbers of vertices and edges in the new G_i^C's.

Deletions. Deletions over the SCC-condensed compound graphs G_i^C, on the other hand, result in a decremental maintenance of the SCCs, which requires either storing the original (i.e., uncondensed) compound graphs or organizing the SCCs in a hierarchical manner [19]. In our implementation, we resort to storing the uncondensed compound graphs along with the condensed compound graphs G_i^C, albeit approaches like [19] may be employed for further optimizations.

A deletion of a local edge (u,v) in partition i is processed over the condensed compound graph G_i^C as follows. If the vertices u, v belong to the same SCC, then we expand this SCC into its original edges and reconnect these edges to the remaining SCCs in G_i^C. Moreover, in case of deletions, some of the existing boundaries may not be connected anymore. We identify such pairs of boundaries and communicate these to the other compute nodes. After receiving this list of deleted boundary edges, we reconstruct the local compound graphs G_j^C (for $j \neq i$) analogously to the insertion case. If, on the other hand, the vertices u, v belong to two different SCCs, then we expand both of them.

Here, the worst-case time complexity to maintain the local boundary edges is $\mathcal{O}((|V_i| + |E_i|) \cdot |I_i| \cdot |O_i|)$, which is the same as for rebuilding the local boundary graphs (see Section 3.3.1). The new compound graphs are condensed via SCC computation, whose worst-case time complexity is $\mathcal{O}(n' + m')$, where n' and m' again are the numbers of vertices and edges in the new G_i^C's.

4. EVALUATION

We next present a detailed empirical evaluation of our proposed indexing and updating strategies for DSR queries. Specifically, we compare the following approaches:

- our DSR approach with a static reachability index (coined "*DSR*"), as described in Section 3.3;
- a naïve enumeration of all pairs of source and target vertices (coined "*DSR-Naïve*"), as described in Section 3.1;
- a generalization of the algorithm described by Fan et al. (coined "*DSR-Fan*") [9] to sets of source and target vertices, as described in Section 3.2;
- an implementation of DSR queries in Apache Giraph (coined *"Giraph"*) [1], as depicted in Appendix 8.4.1;
- an implementation of DSR queries in Giraph++[2] (coined *"Giraph++"*) [31], as depicted in Appendix 8.4.2;
- an extended version of Giraph++ with equivalence sets (coined "*Giraph++wEq"*), as depicted in Appendix 8.4.3.

Further, for *"DSR"*, we report the results in combination with the following local reachability indexes:

- a plain depth-first-search (DFS) strategy which requires no additional index structures except for those described in Section 3.3 (coined *"DSR-DFS"*);
- the multi-source-breath-first-search (MS-BFS) algorithm described by Then et al. [30] which also requires no additional index structures except for those described in Section 3.3 (coined *"DSR-MSBFS"*);
- using FERRARI [28] as a local reachability index that is generated on top of the compound graphs described in Section 3.3 (coined *"DSR-FERRARI"*).

Unless stated otherwise, we report the combination of our DSR index with DFS as the default local search strategy.

Datasets. The list of graph collections we consider for our evaluation is shown in Table 1. All the smaller graphs (including the two Live Journal versions) are obtained from the Stanford Snap[3] project. For our evaluation over larger graphs, we used the real-world Freebase[4] and Twitter[5] snapshots. In addition, we also used the widely popular synthetic LUBM RDF benchmark, which we generated using the UBA 1.7[6] data generator.

Small Graphs	|V|	|E|	Large Graphs	|V|	|E|
Amazon	0.4M	3.3M	LiveJ-68M	4.8M	68.9M
BerkStan	0.7M	7.6M	Twitter-1.4B	41.7M	1,468.4M
Google	0.9M	5.1M	Freebase-500M	97.3M	499.9M
NotreDame	0.3M	1.5M	Freebase-1B	156.6M	999.9M
Stanford	0.3M	2.3M	LUBM-500M	115.6M	500.0M
LiveJ-20M	2.5M	20.0M	LUBM-1B	222.2M	961.4M

Table 1: Graph datasets and sizes

General Setup. We implemented the DSR variants in C++ using GCC-4.7.2 with Boost-1.55 and compiled with -O3 optimization. We used MPICH2-1.4.1 for communication among the compute nodes, using a cluster of 10 nodes which are connected via a 10GBit LAN. Each node has 64GB of RAM and an Intel X5650@2.67GHz quadcore CPU with HT enabled. Giraph and its variants are implemented in Java, where we used Hadoop v0.20 for running Giraph [1]. Appendix 8.4 depicts our actual implementation of DSR queries for the three Giraph variants.

[2]https://issues.apache.org/jira/browse/GIRAPH-818
[3]http://snap.stanford.edu
[4]http://freebase.com
[5]http://an.kaist.ac.kr/traces/WWW2010.html
[6]http://swat.cse.lehigh.edu/projects/lubm/

Graphs	DSR Compound graph Original (#edges)	DSR Compound graph DAG (#edges)	DSR Compound graph Size (MB)	DSR-Fan Dep.graph (#edges)	DSR-Naïve Dep.graph (#edges)
Amazon	1.0M	34.7K	206	622.3M	622.2M
BerkStan	2.1M	0.5M	383	2.2M	2.1M
Google	1.2M	0.2M	302	43.6M	43.6M
NotreDame	0.8M	68.9K	123	4.7M	4.7M
Stanford	0.8M	41.2K	122	1.2M	1.2M
LiveJ-20M	13.7M	1.0M	1,553	861.4M	n/a
LiveJ-68M	44.1M	0.3M	928	n/a	n/a
Freebase-1B	460.4M	241.6M	64,141	n/a	n/a
Twitter-1.4B	1,285.0M	8.2M	20,053	n/a	n/a
LUBM-1B	891.8M	891.3M	107,608	n/a	n/a

Table 2: Index sizes for DSR variants

4.1 Efficiency

For this experiment, we considered several real-world data graphs (both small and large) and the synthetic LUBM graph. We fixed the compute cluster to 6 nodes (i.e., to 5 slaves and 1 master). We randomly selected 10 source and 10 target vertices from all datasets (except LUBM-1B) as queries, thus resulting in 100 reachability comparisons. For LUBM-1B, which is very sparsely connected, we randomly chose 1,000 sources and 1,000 targets, of which only 131 pairs turned out to be reachable.

Table 2 shows the maximum (i.e., per node) uncondensed ("*Original*") and SCC-condensed ("*DAG*") compound-graph sizes as well as the total byte size ("*Size*") for our DSR index in comparison to the dependency-graph sizes for DSR-Fan and DSR-Naïve. In DSR-Fan, for a given DSR query $S \rightsquigarrow T$, all of S and T are used "at once" to generate the dependency graph. In DSR-Naïve, which generates the dependency graph per (s, t) pair, the sizes represented in Table 2 are the average dependency-graph sizes over 100 pairs. SCC compression, which is not feasible for the dynamically generated dependency graph, drastically reduces the sizes of the compound graphs stored at each slave. For example, for the Twitter-1.4B graph, which is highly connected, the size of each compound graph stored at the slaves initially is comparable to the size of the original graph. Applying SCC compression condenses these graphs by a factor of about 150. Also for LiveJ-68M, the SCC compression leads to a much smaller DAG size than for LiveJ-20M, such that our query times are actually lower for LiveJ-68M than for LiveJ-20M.

Table 3 shows our query-processing results. For both the small and large graphs, our approach clearly demonstrates efficiency improvements of several orders of magnitude when compared to the three Giraph variants as well as to DSR-Fan and DSR-Naïve. Even with a single round of communication, DSR-Fan and DSR-Naïve exhibit a considerable overhead in generating the dynamic dependency graph for each query. Specifically, we observed that for LiveJ-20M, DSR-Fan generates a dependency graph of about 861 million edges even when the data graph is partitioned by METIS [17] in order to minimize the cut. Our DSR approach, which benefits from the optimizations we apply when constructing the compound graphs, avoids the repeated generation of a large dependency graph at the master node and therefore is able to achieve very significant performance gains over DSR-Fan and DSR-Naïve. Giraph++ and Giraph++wEq, on the other hand, perform better than the native Giraph implementation, as the former benefit from their local updates of neighboring vertices. This drastically reduced the number of supersteps required for processing a set-reachability query. The equivalence-sets optimization for Giraph++wEq further reduced the communication but only marginally improved the query processing times.

Graphs	Indexing Time DSR	Query Size |S|	|T|	Query Time DSR	Giraph++	Giraph++wEq	Giraph	DSR-Fan	DSR-Naïve
(a) Small Graphs (times in seconds)									
Amazon	2.380	10	10	**0.008**	12.250	11.348	55.034	72.111	855.159
BerkStan	3.048	10	10	**0.009**	44.180	5.680	779.006	2.219	38.036
Google	3.194	10	10	**0.060**	60.154	11.426	53.614	25.210	114.078
NotreDame	1.089	10	10	**0.057**	11.085	12.320	94.787	1.800	50.598
Stanford	1.511	10	10	**0.008**	7.808	8.922	341.976	0.468	6.211
LiveJ-20M	44.536	10	10	**0.227**	19.888	19.262	28.075	521.569	n/a
(b) Large Graphs (times in seconds)									
LiveJ-68M	144.981	10	10	**0.090**	64.728	61.940	93.253	n/a	n/a
Freebase-1B	1,938.670	10	10	**67.849**	1,371.423	1,014.442	1,857.124	n/a	n/a
Twitter-1.4B	6,963.730	10	10	**1.119**	3,065.483	3,046.450	n/a	n/a	n/a
LUBM-1B	2,083.190	1,000	1,000	**1.340**	146.864	142.142	154.407	n/a	n/a

Table 3: Efficiency evaluation (indexing and query times) of DSR approaches for small and large graphs

4.2 Scalability

Next, we evaluated our approach in comparison to the Giraph variants under both strong and weak scaling. We dropped DSR-Fan and DSR-Naïve from these comparisons, as they could not scale to the larger graphs anymore. We considered LiveJ, Freebase, Twitter and the synthetic LUBM graph for our scalability evaluation. We used METIS to partition the graphs and distributed the partitions to up to 10 compute nodes (one of which used as master), and we considered 10 random source and 10 random target vertices as queries. Figures 5(d)(h)(l)(p) also show the robustness of our approach with respect to larger query sets.

Live Journal: Figures 5(a)-(d) depict the scalability evaluation of our approach and the Giraph variants for LiveJ-68M. Figure 5(a) shows the results for a strong scaling. Here, we can observe that DSR scales very well and performs significantly better than the Giraph variants. We also observe that Giraph++ and Giraph++wEq perform slightly better than Giraph by leveraging the node locality and equivalence-sets optimization, respectively. This observation is confirmed further by Figure 5(b), where Giraph communicates about two orders of magnitude more messages compared to Giraph++and Giraph++wEq. Figure 5(c) shows the weak scalability for the 10 by 10 DSR queries.

Freebase: The scalability results for Freebase-1B are shown in Figures 5(e)-(h). We can observe that our approach scales well on average, even when the graph sometimes is rather unevenly partitioned as a result of using METIS. This uneven partitioning also is the reason for the runtime increase from 7 to 8 slaves. By leveraging node locality in Giraph++ and the equivalence-sets optimization in Giraph++wEq, both approaches continue to show performance gains over Giraph, but this time with a more visible difference in the communication costs among the three variants as shown in Figure 5(f). Figure 5(g) shows the weak scalability.

Twitter: We next performed a similar scalability evaluation over Twitter, consisting of more than 1.4 billion edges and more than 41 million vertices. METIS this time resulted in a very skewed partitioning, with one partition containing almost half of the edges and almost one third of the edges being cut edges. This constituted a challenge for our approach, because we compute the boundary graph along with the cut edges. However, since vertices in Twitter are densely connected, the resulting compound graphs at all slaves can very well be condensed using SCC compression, which led to very small graph indexes at the slaves (with a compression factor of more than 150). Without this optimization, Giraph was able to load the Twitter graph but failed to process the set-reachability query, returning an "out of memory" exception. The strong and weak scalability of our approach and the Giraph variants are shown in Figures 5(i) and 5(k).

LUBM: As the final scalability experiment, we considered the synthetic LUBM-1B dataset whose results are shown Figures 5(m)-5(p). Most of the RDF-based LUBM graph is acyclic and sparsely connected. Thus, our SCC condensation for the compound graphs has very low effect on the overall query processing. Figure 5(m) shows the strong scalability of our approach versus the Giraph variants. Figure 5(n) shows the communication costs for different variants of Giraph and our approach. Again, our DSR approach, which evaluates a set-reachability query in a single round of communication, exchanges a very low amount of messages compared to the iterative Giraph variants.

4.3 Updates

We considered the six smaller graphs plus the LiveJ-68M dataset (see Table 1) for updates. We distinguish two principal kinds of incremental update workloads, which we call *bulk updates* and *progressive updates*, respectively.

- For bulk insertions, we start with 60% of randomly chosen edges of the original graph and then increment the graph by 5% of the remaining edges, until we reach the original graph. For bulk deletions, we start with the original graph and decrement the graph in 5% steps.
- For progressive insertions, we randomly pick x% (say, 5%) of edges from the original graph and measure the time to insert these into an index built over the remaining $(100 - x)$% (say, 95%) of edges. We increment x in 5% steps. For progressive deletions, we decrement the original graph by a progressive amount of edges.

We used the same queries as described in Section 4.1 to measure the effect of these update steps on the query times.

Insertions. Figures 6(a)(e) show the update and respective query times for our bulk insertions. It can be observed that the time needed for bulk insertions remains almost constant for each 5% step. Query performance, which depends on the final DAG size, however varied considerably with each update. Next, we considered progressive insertions. From Figure 6(b), it can be clearly seen that the update times are only a fraction of the total rebuild time (see Table 2). Query performance, shown in Figure 6(f), increased marginally at each step, as expected, although also this depends on the final DAG size as a result of the update operation.

Deletions. Deletions are generally more costly in our setting and took almost the same time as building the index from scratch (see Table 2) for both bulk and progressive updates. Figures 6(c)(g) depict the update and respective query times for bulk deletions. While deletion times show a downward trend, query times tend to increase as the graphs become more sparsely connected, thus leading to larger DAG sizes. This is especially visible for the LiveJ-68M dataset. For the case of progressive deletions, as shown

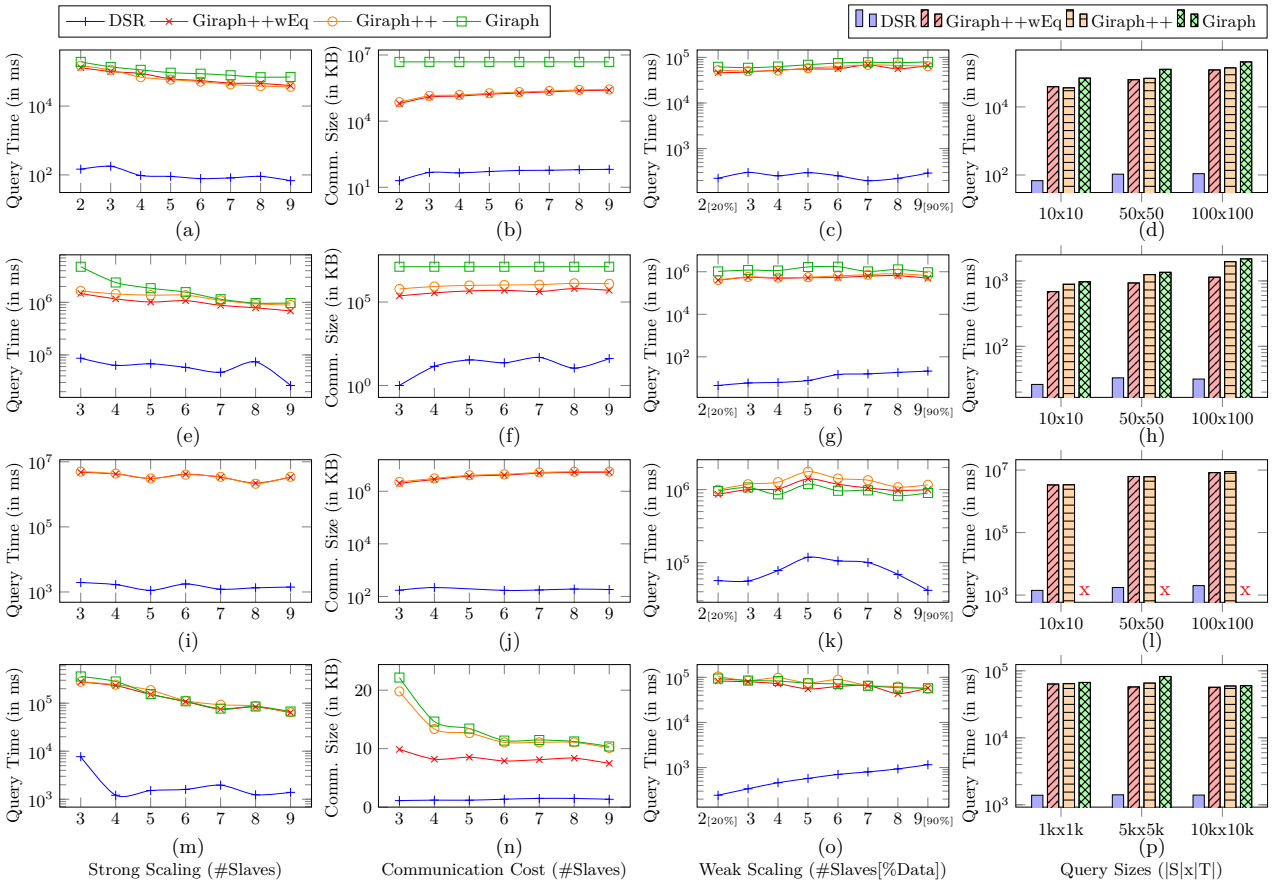


Figure 5: Scalability evaluation for LiveJ-68M (a-d), Freebase-1B (e-h), Twitter-1.4B (i-l), LUBM-1B (m-p)

in Figures 6(d)(h), we observe similar trends in terms of update and query times.

4.4 Parameters

A. Local Reachability Indexes. We next measured our DSR approach in conjunction with three centralized strategies. For all three cases, we condense the local compound graphs via computing SCCs. DSR-DFS uses a standard DFS strategy [6] for processing a DSR query, where no additional index is built over the compound graphs. For each source s and target t in the given DSR query, we perform a DFS to evaluate the reachability $s \rightsquigarrow t$. MS-BFS [30] caches, for each vertex v that is visited during a graph traversal, the reachability of v to all its targets. Thus, if v is another source in the query, we avoid recomputing the reachability and thus save a graph traversal. FERRARI [28] finally provides a tunable tradeoff between index size and query performance. We set both the number of intervals per vertex and the number of seed vertices to 1,000.

Figure 7 shows the effects on query performance when using different local search strategies. For this experiment, we again considered 10 nodes of which one was the master node. We used two real-world datasets, LiveJ-68M and Freebase-1B, for this evaluation. We considered different query sizes to demonstrate the strengths and weaknesses of the three approaches. Figure 7(a) shows the results for LiveJ-68M. We can observe that DSR-DFS takes longer compared to the other two baselines as it requires one graph traversal (in the worst case) for each source. DSR-FERRARI, with its compact reachability index, demonstrates significant performance gains over the other two baselines for different query sizes. On the other hand, for large query sizes, the DSR-MSBFS approach benefits from its memoization and less redundant graph traversal and tends to close the gap to FERRARI. The three strategies show similar trends also for the larger Freebase-1B dataset (see Figure 7(b)).

B. Equivalence-Sets Optimization. By computing equivalence sets among in- and out-boundaries, we are able to reduce both the boundary-graph sizes as well as the number of reachability computations required per slave. Table 4 shows the benefits of this optimization. Figure 8 shows a comparison for the query performance and communication costs with and without equivalence sets in Giraph.

	Query Time *(times in sec.)*		Boundary-Graph Sizes *(#forward; #backward)*	
Graph	**Non-Opt.**	**Opt.**	**Non-Opt.**	**Opt.**
Amazon	0.101	0.008	900; 530	18; 5
BerkStan	0.157	0.110	20,750; 49,462	3,916; 4,981
Google	1.416	1.003	47,822; 98,955	3,759; 6,287
NotreDame	1.085	0.768	16,771; 6,899	2,481; 37
Stanford	0.061	0.038	5,411; 13,942	183; 475

Table 4: Equivalence-sets optimization in DSR

C. Partitioning Strategy. We next considered the effect of the partitioning strategy on the performance of our approach. For this, we used two partitioning strategies: a random hash-partitioning and METIS [17]. We used a cluster of 6 nodes (one of which was dedicated as the master) and evaluated the strategies using a set-reachability query with 10 sources and 10 targets. Table 5 shows the performance comparison of the two partitioning strategies over several real-world graphs. It can be clearly observed that the choice of the partitioning strategy influences the performance of our approach. Hash partitioning (i.e., "random

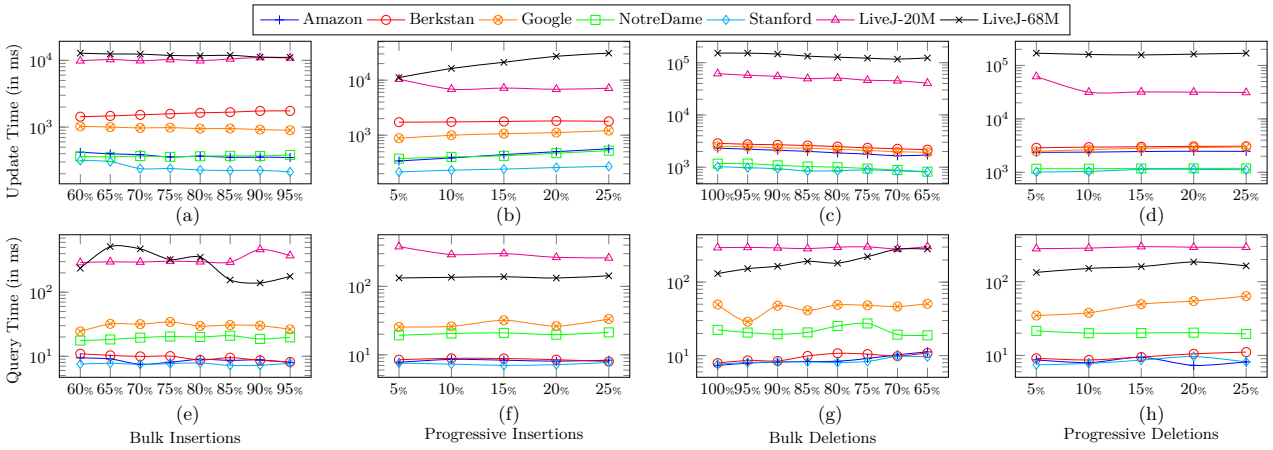


Figure 6: Update evaluation (both insertions and deletions) for various graph collections

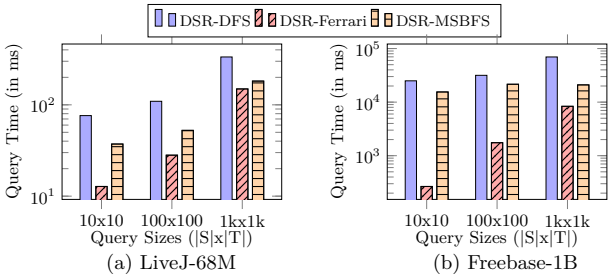


Figure 7: Comparison of local reachability indexes

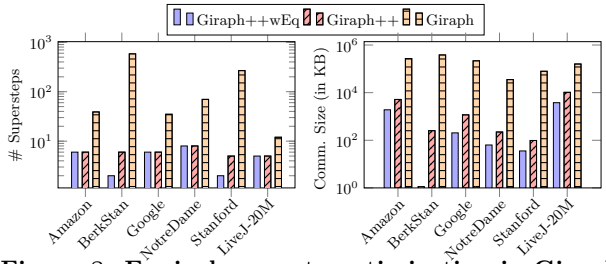


Figure 8: Equivalence-sets optimization in Giraph

sharding") usually results in a drastic increase of cut edges and thus in a lower query performance. METIS partitioning in turn helps in minimizing this cut, which significantly improves the query performance.

	Partitioning *(query times in sec.)*	
Graph	**Hash**	**METIS**
Amazon	0.009	0.008
BerkStan	0.016	0.009
Google	0.330	0.060

	Partitioning *(query times in sec.)*	
Graph	**Hash**	**METIS**
NotreDame	0.085	0.057
Stanford	0.009	0.008
LiveJ-20	0.524	0.227
LiveJ-68	0.188	0.090

Table 5: Impact of hash vs. METIS partitioning

4.5 Applications

A. SPARQL 1.1 with Property Paths. For this experiment, we considered the LUBM-500M and Freebase-500M datasets (both in RDF format). We augmented a distributed RDF store [15] with our DSR approach by modifying its query processor to handle property paths via our new index structures. To evaluate the performance of our approach in processing SPARQL 1.1 queries, we compared against the commercial Virtuoso RDF store [8]. The results are shown in Table 6, while the customized SPARQL queries we used for this evaluation are depicted in Appendix 8.3

B. Social-Network Communities. As another DSR application, we detected connectivities among communities in a social network. This problem is a basic step in many graph-analytics tasks. That is, given two communities C_1 and C_2 together with a set of representative members for each community $S \subseteq C_1$, $T \subseteq C_2$, find all pairs s, t, with $s \in S$ and $t \in T$, such that $s \rightsquigarrow t$. We considered two social network datasets, LiveJ-68M and Twitter-1.4B, for this experiment. We employed the iterative community-detection algorithm by Blondel et al. [3] to identify communities. We then randomly picked two communities, and from each we picked 10 to 1,000 members as representatives. We then ran our DSR approach to identify all reachable pairs among these representatives. The results are shown in Table 7.

(a) LUBM-500M *(query times in sec.)*					**Geo.-**
	#Slaves	**L1**	**L2**	**L3**	**Mean**
DSR	1	6.437	0.331	42.681	4.497
DSR	5	**1.250**	**0.162**	**8.516**	**1.199**
Virtuoso (cold)	1	10.050	12.624	57.776	19.425
Virtuoso (warm)	1	4.963	5.452	56.603	11.527

(b) Freebase-500M *(query times in sec.)*					**Geo.-**
	#Slaves	**F1**	**F2**	**F3**	**Mean**
DSR	1	1.084	1.568	0.677	1.048
DSR	5	**0.356**	0.642	**0.423**	0.459
Virtuoso (cold)	1	6.590	4.112	13.809	7.206
Virtuoso (warm)	1	1.196	**0.002**	5.601	**0.238**

Table 6: SPARQL 1.1 queries with property paths

	LiveJ-68M *#Communities: 5,032*		**Twitter-1.4B** *#Communities: 17,121*	
Query Size (|S|x|T|)	**Query Time** *(in sec.)*	**#Pairs**	**Query Time** *(in sec.)*	**#Pairs**
10x10	0.065	81	1.339	63
100x100	0.164	8,184	2.476	8,526
1kx1k	0.717	784,947	10.175	712,725

Table 7: Community connectedness using DSR

4.6 Summary of Results

Our experiments confirm the significantly improved efficiency (with a gain in query times of several orders of magnitude) of our DSR index compared to iterative approaches such as Apache Giraph and variants of [9]. Moreover, we are also able to demonstrate the good update support of our index structure, which—in particular for insertions—behaves much better in practice than suggested by the worst-case bounds we provide in Section 3.3.3. We believe that insertions are the much more likely use-case for managing large, dynamic graphs (e.g., Twitter streams), while deletions, which are costly to handle for any kind of graph-compression technique, are much more uncommon in practice. Also, there is also hardly any support for updates in the centralized approaches (such as [28]), which restricts our local search strategy to a simple DFS or BFS in this case.

Further experiments demonstrate the robustness of our approach under different parameters and show its viability for various large-scale graph-analytics tasks.

5. RELATED WORK DISCUSSION

Centralized Approaches. The reachability problem in directed graphs is one of the most fundamental graph problems and thus has been tackled by a plethora of *centralized indexing techniques* [5, 12, 16, 18, 25, 28, 32, 33, 34, 36] (just to name a number of recent approaches). All of these aim to find tradeoffs among query time and indexing space which, for a directed graph $G(V, E)$, are in between $\mathcal{O}(|V|+|E|)$ for both query time and space consumption when no indexes are used, and $\mathcal{O}(1)$ query time and $\mathcal{O}(|V|^2)$ space consumption when the transitive closure is fully materialized.

Recently, Gao et al. [12] proposed a suitable, but centralized indexing strategy, based on a notion of *equivalence sets* of graph vertices that have the same reachability properties. [30], on the other hand, focused on the *query-time optimization* of multi-source BFS searches. However, there exist hardly any works so far on distributed reachability queries [9]. Fan et al. [9] recently discussed distributed, but single-source, single-target reachability algorithms for partitioned graphs and also provide performance guarantees. For a directed graph and given cut, [9] uses a partially iterative and partially indexing-based evaluation to find the reachability for a given pair of vertices over a classical master-slave architecture. In Section 3, we therefore provide a detailed review of the techniques proposed in [9], while the query-time processing we perform based on equivalence sets to a large extent resembles also the techniques described in [12, 30] for a centralized setting. However, unlike in [12], we do not enumerate the actual path sequences.

Distributed Graph Engines. Distributed graph engines such as Pregel [22], GraphX [35], GraphLab [20], Trinity [29], PowerGraph [21], Giraph [1] and Giraph++ [31] are either based on MapReduce [22, 35, 21], or they implement their own, proprietary communication protocols via Message Passing [13, 20, 29]. Giraph, for example, offers the `sendMessage(.)` and `compute(.)` methods as generic API functions to implement various kinds of graph algorithms (including BFS and DFS). To implement a *single-source, single-target reachability query* over a directed graph, each iteration over the `compute(.)` method (as it is required for a single BFS/DFS step), however, results in a new call of the Map function or so-called "superstep". Among two such supersteps, messages are communicated among all compute nodes, which is a strategy that—due to the a-priori unknown amount of iterations—usually does not permit for interactive query response times. For *multi-source, multi-target queries*, on the other hand, this approach scales well with the query size due to the possibility to implement shared computations in the `compute(.)` method.

A similar observation holds for GraphLab [21], Trinity [29] and PowerGraph [13] which implement asynchronous protocols based on the Message Passing Interface (MPI) [11]. PowerGraph, for example, which is specifically tuned for skewed graphs, implements a judiciously chosen schedule of exchanged messages, but also here the worst-case amount of iterations remains equal to the diameter of the graph. The very recently proposed Giraph++ [31] framework (built on top of Giraph) provides further optimizations by shifting from a purely node-centric to a more graph-centric ("think like a graph") compute paradigm. All (local) messages among the vertices within the same graph partition are performed inside a superstep, while other messages are processed only between two such supersteps. This significantly improves the performance by minimizing the number of messages but also gives a much higher degree of freedom in the implementation of various graph algorithms.

Thus, on the one hand, the generic abstraction layers of these distributed graph engines make it difficult to exploit shared computations and yet to fully benefit from the underlying distribution scheme. On the other hand, these engines generally do not support graph-indexing techniques known from the centralized approaches, which could ideally be employed to even completely avoid iterative communication rounds among the compute nodes. Since Giraph++ offers the most flexible API among the aforedescribed engines, we extensively compared our approach against two principle implementations of DSR queries in the very recent Giraph++ framework (including one native Giraph version).

Processing SPARQL 1.1 Property Paths. Finally, combining relational joins with reachability predicates in SPARQL 1.1 inherently leads to a multi-source, multi-target (i.e., set-) reachability problem. Very few works so far focused on the combined processing of relational joins with additional graph-reachability predicates [4, 10, 14, 27]. In earlier works, the bisimulation-based indexing of path expressions for XML trees [23] has been extended to RDF graphs [24], but the latter did not yet consider property paths. Likewise, [4] proposed an index structure that is limited to DAGs obtained from XML/XLink(s). [26] finally investigated an initial approach to evaluate property paths via MapReduce. Also here, we are aware of only one RDF engine that is available for processing property paths in SPARQL 1.1, namely Virtuoso [8], against which we compared our approach experimentally.

6. CONCLUSIONS

We investigated a generalized form of the well-known reachability problem in directed graphs, which (1) considers both sets of source and target vertices as queries, and (2) allows the underlying graph to be partitioned and hence be distributed across multiple compute nodes. The DSR problem is a basic building block and thus has a plethora of applications in graph analytics and query-processing tasks. As our core contribution, we presented an efficient and scalable framework for processing DSR queries and also studied its formal properties. By precomputing and materializing the reachability information among vertices along the cut of a partitioned data graph, our approach is guaranteed to require at most one round of communication among the compute nodes to resolve any DSR query. Our approach exhibits a very good support for incremental vertex and edge insertions, while our current implementation resorts to just a basic support for respective deletions. Moreover, any state-of-the-art centralized reachability index may be applied to the local graph partitions to further accelerate query-processing times. Our evaluation over both real-world and synthetic graphs and in comparison to very recent iterative approaches also empirically demonstrated the viability of our approach. As a future work, we aim to explore more types of reachability problems with additional length restrictions and regular expressions over path labels as constraints.

7. REFERENCES

[1] http://giraph.apache.org/.

[2] http://www.w3.org/TR/sparql11-property-paths/.

[3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[4] J. Cheng, J. X. Yu, and B. Ding. Cost-Based Query Optimization for Multi Reachability Joins. In *DASFAA*, pages 18–30, 2007.

[5] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[7] C. Demetrescu and G. F. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms*, 4(3):353 – 383, 2006.

[8] O. Erling and I. Mikhailov. Virtuoso: RDF Support in a Native RDBMS. In *SWIM*, pages 501–519, 2009.

[9] W. Fan, X. Wang, and Y. Wu. Performance guarantees for distributed reachability queries. *PVLDB*, 5(11):1304–1315, 2012.

[10] W. Fan, X. Wang, and Y. Wu. Answering graph pattern queries using views. In *ICDE*, pages 184–195, 2014.

[11] T. M. Forum. MPI: A Message Passing Interface, 1993.

[12] S. Gao and K. Anyanwu. PrefixSolve: efficiently solving multi-source multi-destination path queries on RDF graphs by sharing suffix computations. In *WWW*, pages 423–434, 2013.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, pages 17–30, 2012.

[14] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling Kleene: fast property paths in RDF-3X. In *GRADES*, pages 14:1–14:7, 2013.

[15] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *SIGMOD*, pages 289–300, 2014.

[16] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS*, 36(1):7, 2011.

[17] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[18] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, pages 31–46, 2012.

[19] J. Łącki. Improved deterministic algorithms for decremental reachability and strongly connected components. *ACM Trans. Algorithms*, 9(3):27:1–27:15, 2013.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*, pages 340–349, 2010.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 5(8):716–727, 2012.

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

[23] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[24] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, pages 406–421, 2012.

[25] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan. Managing large graphs on multi-cores with graph awareness. In *USENIX*, pages 41–52, 2012.

[26] M. Przyjaciel-Zablocki, A. Schätzle, T. Hornung, and G. Lausen. RDFPath: Path query processing on large RDF graphs with MapReduce. In *ESWC*, pages 50–64, 2011.

[27] M. Sarwat, S. Elnikety, Y. He, and M. F. Mokbel. Horton+: A distributed system for processing declarative reachability queries over partitioned graphs. *PVLDB*, 6(14):1918–1929, 2013.

[28] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.

[29] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013.

[30] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *PVLDB*, 8(4):449–460, 2014.

[31] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[32] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[33] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

[34] R. R. Veloso, L. Cerf, W. M. Jr., and M. J. Zaki. Reachability queries in very large graphs: A fast refined online search approach. In *EDBT*, pages 511–522, 2014.

[35] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: a resilient distributed graph system on Spark. In *GRADES*, 2013.

[36] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: Scalable Reachability Index for Large Graphs. *PVLDB*, 3(1-2):276–284, 2010.

[37] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. Technical Report UCB/EECS-2010-53, UC Berkeley, 2010.

8. APPENDIX

8.1 Computing Equivalence Sets

Algorithm 3 computes the forward-equivalent sets of vertices in each graph partition G_i as follows. Given a graph partition G_i with in-boundaries I_i, the forward-equivalent sets EQ_i^f are computed as follows. First, the graph is condensed into its DAG representation G_i' by computing the strongly connected components of G_i. Next, the target vertices $S(I_i) - I_i$ are chosen as the successors of the in-boundaries I_i. We define a Boolean array rep[] (for "representative"), whose truth value for a given boundary b_m denotes whether a forward-equivalent set (i.e., an in-virtual vertex) v is formed with any other boundary b_l, for $m > l$. The rep[] array is initially set to "true" for all boundaries. A boundary b_l is equivalent to b_m, iff either b_l and b_m belong to same SCC (Lines 11-14) or have the same reachability set rset (Lines 17-19). rset[j] for the j^{th} boundary denotes the set of vertices from $S(I_i) - I_i$ that are reachable from b_j. The computed equivalence set v starting at boundary b_l, where rep[b_l]:=true, is added to EQ_i^f at the end of each iteration (outer loop). With minor modifications from $S(I_i) - I_i$ to $P(O_i) - O_i$ (thus using predecessors instead of successors), the algorithm can similarly be adapted to compute the backward-equivalent sets EQ_i^b of out-boundaries.

Algorithm 3: Computing Forward-Equivalent Sets

Input: Subgraph G_i, In-boundaries I_i
Output: Forward-equivalent sets EQ_i^f

```
1  EQ_i^f := ∅
2  G_i' := condense(G_i)        ▷ graph condensation via SCC computation
3  S(I_i) := successors(I_i, G_i')
4  rep[1..|I_i|] := true
5  rset[k] := ∅
6  for l = 1 ... |I_i| do
7      if rep[l] then
8          v := {b_l}
9          if rset[b_l] = ∅ then
10             rset := localSetReachability({b_l}, S(I_i) − I_i)
11         for m = l + 1 ... |I_i| do
12             if scc(b_l) = scc(b_m) then
13                 v := v ∪ {b_m}
14                 rep[m] := false
15             else
16                 rset := localSetReachability({b_m}, S(I_i) − I_i)
17                 if rset[b_l] = rset[b_m] then
18                     v := v ∪ {b_m}
19                     rep[m] := false
20         EQ_i^f := EQ_i^f ∪ v
```

8.2 Proofs of Theorems

8.2.1 Proof of Theorem 1

PROOF. Let $P = \{(s, u_1), \ldots, (u_m, u), (u, v), (v, v_n), \ldots, (v_1, t)\}$ denote the set of edges along a path from source s to target t. Then the following holds: edge $(u, v) \in P$, with $u \in V_i, v \in V_j$, can either be (1) a *cut edge* iff $i \neq j$, (2) a *local edge* in partition i iff $i = j$, or (3) a *non-local edge* with respect to partition k iff $i = j$ and $i \neq k$.

Case A: Let all edges in P be either local to partition i (1) or be a cut edge (2) among partitions i, j. Then, from Definition 6 of the compound graph $G_i^C = (V_i^C, E_i^C)$, it follows that $P \subseteq E_i^C$. That is, the reachability $s \rightsquigarrow t$ can be computed entirely locally at partition i using E_i^C.

Case B: Let $(u, v) \in P$ such that (u, v) is a non-local edge (3) to partition i but a local edge (1) to another partition j, with $i \neq j$. That is, $(u, v) \in E_j^C$ but $(u, v) \notin E_i^C$. From this, it follows that $\exists p, q$ such that the edges (u_{p-1}, u_p), $(v_q, v_{q-1}) \in P$ are cut edges (2), where u_p, $v_q \in V_j$ with $1 \leq p \leq m$ and $1 \leq q \leq n$. Next, we choose (u_{p-1}, u_p), $(v_q, v_{q-1}) \in P$ as the edges with the largest indices of p, q for which this property holds. This choice ensures that a path from u_p to v_q via (u, v) resides entirely in partition j. Then, vertex u_p forms an in-boundary while vertex v_p forms an out-boundary of partition j, and the edges of the sub-path $\{(u_{p-1}, u_p), \ldots, (u_m, u), (u, v), (v, v_n), \ldots, (v_q, v_{q-1})\} \subseteq P$ reside in partition j. In this case, by the construction of the boundary graph $G_i^B = (V_i^B, E_i^B)$, we added a reachability edge (u_p, v_q) to E_i^B (see Definition 4). This means, in the optimized E_i^B, we add an edge (v, ν), where $u_p \in v$ is an in-virtual vertex and $v_q \in \nu$ is an out-virtual vertex. Since $E_i^B \subseteq E_i^C$ (see Definition 6), there exists a path $\{(s, u_1), (u_1, u_2), \ldots, (u_{p-1}, v), (v, \nu), (\nu, v_{q-1}), \ldots, (v_1, t)\}$ in partition i, thus again ensuring that the reachability of $s \rightsquigarrow t$ can be computed entirely locally at partition i using E_i^C. □

8.2.2 Proof of Theorem 2

PROOF. The proof is simple and leverages the result of Theorem 1. Let $P = \{(s, u_1), (u_1, u_2), \ldots, (u_{p-1}, u_p), \ldots, (u_n, t)\}$ denote the set of edges along the path from source s to target t, where $s \in V_i, t \in V_j$. If $i \neq j$, then there exists an edge $(u_{p-1}, u_p) \in P$ which is a cut edge (1). That is, $u_{p-1} \in V_k$, $u_p \in V_j$ and $k \neq j$. Next, we choose the largest index p such that the subpath $\{(u_p, u_{p+1}), \ldots, (u_n, t)\}$ resides entirely at partition j. Since (u_{p-1}, u_p) is a cut edge and $u_p \in V_j$, u_p forms an in-boundary of partition j. We next choose the smallest q such that $\{(s, u_1), \ldots, (u_{q-1}, u_q)\}$ resides entirely at partition i. Note that in the optimized boundary graph, we actually use virtual vertices instead of the regular ones, which we omit here for simplicity.

Path P can be thus written as a concatenation of subpaths $P_1 = \{(s, u_1), \ldots, (u_{q-1}, u_q)$, $P_2 = (u_q, u_{q+1}) \ldots (u_{p-1}, u_p\}$, and $P_3 = \{(u_p, u_{p+1}), \ldots, (u_n, u_t)\}$. According to Theorem 1, P_1 and P_3 can be computed entirely at partition i and j, respectively. u_q and u_p thus are an out- and in-boundary of partition i and j, respectively, i.e., $u_q, u_p \in V_C$. As per the construction of the local compound graphs (see Definition 6), P_2 can be evaluated at either partition i or j. Thus, the reachability problem $s \rightsquigarrow t$ is reduced to two reachability problems: (a) $s \rightsquigarrow u_p$ at partition i and (b) $u_p \rightsquigarrow t$ at partition j. To find such a u_p, we iterate over all in-boundaries b of partition j residing at partition i. We then compute the reachability $s \rightsquigarrow b$ and communicate the reachable in-boundaries to partition j. Thus, answering $s \rightsquigarrow t$, where $s \in V_i$ and $t \in V_j$, with $i \neq j$, requires a local processing at two partitions i, j and involves only a single step of message exchange from partition i to partition j. □

8.3 SPARQL 1.1 Queries with Property Paths

A. LUBM Queries

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
L1: SELECT * WHERE { ?x rdf:type ub:ResearchGroup .
     ?x ub:subOrganizationOf* ?y. ?y rdf:type ub:University. }
```

L2: SELECT * WHERE { ?x rdf:type ub:FullProfessor. ?x ub:headOf ?d. ?d ub:subOrganizationOf* ?y. ?y rdf:type ub:University.}

L3: SELECT * WHERE { ?r1 rdf:type ub:ResearchGroup . ?r1 ub:subOrganizationOf* ?y. ?y rdf:type ub:University . ?r2 rdf:type ub:ResearchGroup. ?r2 ub:subOrganizationOf* ?y.}

B. Freebase Queries

@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix fb: <http://rdf.freebase.com/ns>

F1: SELECT * WHERE { ?p fb:people.person.place_of_birth ?city . ?city fb:location.location.containedby* ?state. ?country fb:location.location.contains ?state. }

F2: SELECT * WHERE { ?p fb:people.person.place_of_birth ?city . ?city fb:location.location.containedby* ?state. ?country fb:location.location.contains ?state. ?p fb:award.award _winner.awards_won ?prize. ?p rdf:type fb:government.us _president. }

F3: SELECT * WHERE {?p fb:award.award_winner.awards_won ?prize. ?prize rdf:type* ?z . ?z fb:award.award_honor.ceremony> ?c. ?p fb:people.person.sibling_s* ?p1. ?p1 fb:award.award_winner.awards_won> ?prize. }

8.4 Giraph Implementations of DSR Queries

8.4.1 Giraph

The following program code illustrates our implementation of DSR queries in Giraph. In superstep 0, all source vertices are first added to a newSources array. Thus, the function isSource(.) returns true if the vertex is a source. In the subsequent supersteps, newSources represents the additional sources from which the current vertex v is reachable. If newSources is not empty, then we iteratively propagate these sources to all neighbors of vertex v.

Distributed Set Reachability in Giraph

```
public void compute(Vertex v, Iterable m){
  ArrayList<Integer> newSources = new ArrayList<Integer>();
  if(getSuperStep() == 0){
     if(isSource(v))
       newSources.add(v.getId().get());
     v.getValue().clearSources()
  }else
    for(IntWritable msg : m)
      newSources.add(m.get());

  newSources.removeAll(v.getValue().getSources());
  if(newSources.size() > 0){
      v.addSources(newSources);
      for(Edge<IntWritable, NullWritable> e : v.getEdges()){
        IntWritable nb = e.getTargetVertexId();
        for(int src: newSources)
          sendMessage(nb,new IntWritable(src));
      }
    }
  }
}
```

8.4.2 Giraph++

Unlike in Graph, the Giraph++ API exposes the underlying partitioning information along with each call of the compute(.) function. The code for DSR processing is similar to Giraph, except that the vertices that are local to the current source vertices are directly updated using a centralized local reachability computation via localProcess(.). After the local processing, for each vertex we again communicate its reachable list of vertices to the remote neighbors.

Distributed Set Reachability in Giraph++

```
public void compute(Partition p){
   ArrayList<Integer> q_sources = new ArrayList<Integer>();
   ArrayList<Integer> newSources = new ArrayList<Integer>();
   if(getSuperStep() == 0){
     if(isSource(v))
       sources.add(v.getId().get();
   }else{
     MessageStore<IntWritable, IntWritable> mstore
           = getCurrentMessageStore();
     for(Vertex v : p.getVertices()){
       if(mstore.hasMessagesForVertex(v.getId())){
         newSources.clear();
         for(IntWritable message :
           mstore.getVertexMessages(v.getId()))
           newSources.add(message.get());
         newSources.removeAll(v.getValue().getSources());
         if(newSources.size() > 0){
           q_sources.add(v.getId());
           v.getValue().addNewSources(newSources);
         }
       } }
   }
   localProcess(p,q_sources);
   for(Vertex v : p.getVertices()){
     if(v.getValue().getNewSources().size() > 0){
       for(Edge<IntWritable, NullWritable> edge
                  : v.getEdges()){
         int nb = edge.getTargetVertexId().get();
         if(!p.contains(nb)
           for(int src : v.getValue().getNewSources())
             sendMessage(nb,new IntWritable(src));
       }
       v.getValue().addSources(v.getValue().getNewSources());
       v.getValue().getNewSources().clear();}
   }
}
```

8.4.3 Giraph++wEq

The following code depicts our DSR implementation in Giraph++wEq, including our proposed equivalence-sets optimization. We first compute equivalence sets in our DSR system and prepare an adjacency graph as input to Giraph. For each vertex v in the input graph, in addition to its adjacent neighbors, we also add their equivalence sets (our in-virtual vertices) as counterparts. This graph is loaded into Giraph using a custom input reader. The below code shows the DSR computation. The implementation shares a major part of the code with the Giraph++implementation, where the only difference lies in the communication of the reachable sets of vertices in each superstep. After the local processing, we iterate over each vertex and send its reachable list of sources to only the in-virtual vertices instead of all neighbors.

Distributed Set Reachability in Giraph++wEq

```
public void compute(Partition p){
   ArrayList<Integer> q_sources = new ArrayList<Integer>();
   ArrayList<Integer> newSources = new ArrayList<Integer>();
   if(getSuperStep() == 0){
     if(isSource(v))
       sources.add(v.getId().get();
   }else{
     MessageStore<IntWritable, IntWritable> mstore
           = getCurrentMessageStore();
     for(Vertex v : p.getVertices()){
       if(mstore.hasMessagesForVertex(v.getId())){
         newSources.clear();
         for(IntWritable message :
           mstore.getVertexMessages(v.getId()))
           newSources.add(message.get());
         newSources.removeAll(v.getValue().getSources());
         if(newSources.size() > 0){
           q_sources.add(v.getId());
           v.getValue().addNewSources(newSources);
         }
       }
     }
   }
   localProcess(p,q_sources);
   for(Vertex v : p.getVertices()){
     if(v.getValue().getNewSources().size() > 0){
```

```
        for(int eq_nb : v.getEqList())
           for(int src : v.getValue().getNewSources())
         sendMessage(new IntWritable(eq_nb),new IntWritable(src));
        v.getValue().addSources(v.getValue().getNewSources());
        v.getValue().getNewSources().clear();}
    }
}
```