QFrag: Distributed Graph Search via Subgraph Isomorphism

Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos
Qatar Computing Research Institute - HBKU
HBKU Research Complex 1
Doha, Qatar
{mserafini,gmorales,gsiganos}@hbku.edu.qa

ABSTRACT

This paper introduces QFrag, a distributed system for graph search on top of bulk synchronous processing (BSP) systems such as MapReduce and Spark. Searching for patterns in graphs is an important and computationally complex problem. Most current distributed search systems scale to graphs that do not fit in main memory by partitioning the input graph. For analytical queries, however, this approach entails running expensive distributed joins on large intermediate data.

In this paper we explore an alternative approach: replicating the input graph and running independent parallel instances of a sequential graph search algorithm. In principle, this approach leads us to an embarrassingly parallel problem, since workers can complete their tasks in parallel without coordination. However, the skew present in natural graphs makes this problem a *deceitfully* parallel one, i.e., an embarrassingly parallel problem with poor load balancing. We therefore introduce a *task fragmentation* technique that avoids stragglers but at the same time minimizes coordination. Our evaluation shows that QFrag outperforms BSP-based systems by orders of magnitude, and performs similar to asynchronous MPI-based systems on simple queries. Furthermore, it is able to run computationally complex analytical queries that other systems are unable to handle.

CCS CONCEPTS

• **Information systems → Data analytics**; • **Computing methodologies → Distributed algorithms**;

KEYWORDS

Graph Search, Load Balancing, Bulk Synchronous Processing

ACM Reference format:
Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. QFrag: Distributed Graph Search via Subgraph Isomorphism. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 15 pages.
DOI: 10.1145/3127479.3131625

1 INTRODUCTION

Search is a fundamental primitive in every database. It is especially important in graph databases, in which patterns of interest might be hard to find manually. Informally, graph search implies executing a *query* that specifies a *pattern* of interest in the input graph. The result of the query are *subgraphs* of the input graph that match the pattern. This problem is known in the literature as *subgraph isomorphism*.

A large number of systems handle graphs and offer search capabilities: graph databases such as Neo4j,[1] many RDF stores [1, 16, 18, 26, 28], and distributed frameworks on Apache Spark such as GraphFrames.[2] However, many of these systems are optimized to run *transactional* queries with high selectivity. For example, the popular LUBM benchmark for RDF search [14] mainly consists of queries that start from a specific vertex and select a small number of vertices up to few hops away (e.g., 'Find all students who take CS-101 at MIT'). This kind of query has a small intermediate state and, in most cases, it can easily be answered by single-server systems such as Neo4j and RDF-3X [28]. The few distributed systems among the existing solutions focus on scaling the same type of queries to large graphs that do not fit in the memory of a single server [1, 37].

While this problem has its own merit, there is currently a lack of graph querying systems that can deal with lower selectivity *analytical* queries. These queries are usually part of a data analysis pipeline in the context of complex graph analysis and mining workflows. Consider for example, queries that do not start from a specific vertex, such as 'Find pairs of students who attended the same course but are not friends', or 'Find groups of three friends who studied at three different universities'. These queries have much lower selectivity than the previous example because *any* university and course could be involved in a match.

Efficiently running complex analytical graph queries is becoming more and more important. Recent work proposes graphs as a formalism to represent dependencies and structures in very large, unstructured datasets, which are sometimes called "data lakes" or "data oceans" [12, 25]. For example, Mavlyutov et al. [25] introduce the concept of Dependency-Driven Analytics (DDA), where raw data is mapped into a dependency graph, following some user-defined rules, and then the graph can be explored to navigate the data in a more understandable and organized way. When examining the system log of a big data system, for example,

[1] http://neo4j.com
[2] http://graphframes.github.io

vertices may represent the execution of a specific task t at a given time, a job j, or a server s, and edges can express that task t is part of job j, or that a specific execution of t has run on server s. Complex analytical queries such as "*find all pairs of jobs having a task that failed while running on the same server*" can be useful in debugging. Guider, a DDA tool used by Microsoft, aggregates multiple system logs into a graph that fits into the main memory of a single server, and uses a standalone Neo4j instance to query it. But as we have discussed previously, Neo4j is not optimized for complex analytical queries, which may thus require bypassing Neo4j and directly sifting through the raw logs.

Despite their apparent simplicity, analytical queries such as the ones presented above can quickly become computationally intractable, even on small graphs. Our evaluation shows that existing systems are not able to deal with analytical queries involving patterns of as few as three or four vertices, even on relatively small graphs with one million vertices, which easily fit in the memory of one server. This result is not surprising, given that the subgraph isomorphism problem is NP-Complete [34]. Partitioning the input graph introduces an additional communication bottleneck to an already computationally complex problem.

Our proposal. In this paper we introduce QFrag, a framework for distributed graph search that is specifically designed to deal with computationally complex queries rather than with extremely large graphs. QFrag is based on the insight that graph search is inherently computationally expensive, and that many practical graphs fit the (ever-growing) main memory of a single server, as also observed more in general for data analytics workloads [32]. Therefore, the design principle of QFrag is to *distribute the computation, not the data*. As such, the input graph is replicated on several servers.

Replicating the input graph allows QFrag to reuse the decades of research in sequential algorithms for subgraph isomorphism. A simple approach to parallelize graph search is to have multiple workers run a sequential pattern matching algorithm in parallel, with each worker starting on a different set of vertices. With this approach, graph search becomes an embarrassingly parallel problem, as no coordination among workers is required. However, while simple, this approach is not efficient due to the skew present in natural graphs. The overall running time is often clearly dominated by a handful of workers, thus limiting the gains from increased parallelism. We call this class of problems *deceitfully parallel*: problems that are embarrassingly parallel in principle, but exhibit poor scalability due to load imbalance and stragglers.

Parallelization in QFrag. QFrag uses a *task fragmentation* approach to deal with deceitfully parallel problems. The technique consists in subdividing a sequential task into a sequence of sequential subtasks. In the case of graph search, the sequential task matches a query pattern starting from a specific vertex in the input graph by using a sequential subgraph isomorphism algorithm. Without task fragmentation, the system runs K instances of the sequential tasks in parallel, where K is the number of workers, in one single superstep. With task fragmentation, each task is split into H subtasks, executed over H supersteps. The goal of task fragmentation is to ensure that the execution times for subtasks in each superstep are as uniform as possible. To this end, if the work associated with a subtask is above average, task fragmentation redistributes some of it across all workers, and executes it in the next superstep.

An important design choice of QFrag is that it runs on top of bulk synchronous parallel (BSP) systems such as MapReduce [11] and Spark [41]. QFrag can thus be easily integrated in data analytics pipelines running on one of these platforms: for example, a pipeline can use a tool for large-scale SQL-like queries to build a graph, then QFrag to filter subgraphs matching a pattern of interest, and finally run a user-defined function to analyze the subgraphs. Developing such integrated pipelines is much needed by the industry, as also reported by the authors of Guider [25]. The challenge with running on top of BSP systems is the need to minimize coordination, which is required for load redistribution. The replication of the input graph and task fragmentation are key design choices to achieve this goal.

Thanks to its design, QFrag is faster than production-grade systems such as Neo4j and GraphFrames (based on Spark). It outperforms these systems by up to 2 and 3 orders of magnitude, respectively, when running sequentially. The efficiency of QFrag is such that it runs more than one order of magnitude faster than GraphFrames on 320 workers, even when running sequentially. QFrag matches the speed of asynchronous, MPI-based systems such as TriAD on simple queries, while at the same time being able to scale to more complex queries, which any other system is unable to handle. This is because existing systems like GraphFrames and TriAD scale out by partitioning the input graph, which leads to additional coordination costs, and because task fragmentation further improves the performance of QFrag by up to *four* times.

Clearly, this design philosophy results in some limitations in the system. First and foremost, QFrag cannot process graphs which do not fit in main memory. Nevertheless, in our experiments we show that the computational limits are reached long before memory becomes an issue. In addition, QFrag is geared towards analytical workloads, rather than transactional ones. The design assumes a read-only dataset and long-running analytical queries, and is thus not suited for high-throughput low-latency transactional-style queries.

Our contributions can be summarized as follows:

- we introduce QFrag, a distributed system for graph search based on two main design principles:
  - leveraging subgraph isomorphism rather than joins;
  - distributing the computation rather than the data;
- QFrag parallelizes state-of-the-art algorithms for subgraph isomorphism; as a byproduct, we obtain the first distributed algorithm for subgraph isomorphism;

- QFrag uses a task fragmentation approach to deal with skew in the input graphs, which improves its performance by up to 4× compared to a naïve approach;
- an extensive experimental evaluation shows superior performance than other state-of-the-art distributed graph search systems in running complex analytical queries;

2 TASK FRAGMENTATION

Task fragmentation is a technique to parallelize the execution of *deceitfully parallel* sequential algorithms. It is particularly suited for algorithms running on BSP systems that can observe stragglers.

System model and requirements. We consider systems where workers coordinate via a BSP approach, by which a computation consists of one or more supersteps [40]. In each superstep, a worker processes input messages sent by the other workers in the previous superstep, updates its local state, and sends messages to other workers, if necessary. All workers execute supersteps synchronously: a superstep starts only after all messages from the previous superstep have been delivered. This fact has two important implications for performance. First, workers interrupt computation while they send messages to other workers and wait for their messages. Minimizing the size of the messages maximizes the amount of time spent doing computation. Second, the time taken by each worker to complete its computation should be balanced, lest workers end up being idle waiting for some other straggler. Task fragmentation allows to balance load while minimizing coordination by making sure that workers take approximately the same time to complete a superstep.

We consider a *task* that is executed by a set of K workers, each running on a different server and having some local state. Each worker k receives a set of initial input data items I_k^0 at the beginning of the computation. The following discussion describes the execution of a single BSP superstep, but the technique can be applied to algorithms executing multiple supersteps. For each data item in I_k^0, each worker can execute the sequential algorithm for the superstep from beginning to end, and thus produce a partial output without requiring any coordination with other workers. Task fragmentation introduces additional coordination to avoid stragglers.

Technique. At a high level, task fragmentation breaks the task T into a sequence of subtasks $ST^1, \ldots, ST^H$ to be executed over H supersteps. Each subtask ST^h takes a data item as input and outputs a set of data items, which are intermediate results to be used as input for the next subtask ST^{h+1}. The intermediate results consist of a list of arbitrary data items, similar to a dataflow computation. Fragmentation provides opportunities to balance load by sharing the work associated with the intermediate results.

The intermediate results are split into two groups, *regulars* and *outliers*, according to a cost function that estimates the execution time for the next subtask on the given data item. Processing regular intermediate results is expected to take approximately the same time at each worker, so they are locally processed by the next subtask in the same worker and in the same superstep h. Processing regular items immediately and locally reduces the amount of data that needs to be shuffled over the network. Conversely, outliers are the intermediate results that cause skew in the execution time per worker, so they are split, shuffled to other workers, and processed in the following superstep.

Figure 1 shows task fragmentation for a single worker k. The initial inputs and intermediate results are defined as follows. Superstep h starts with processing the input set M_k^h. For the first subtask, M_k^h is the set of initial inputs I_k^0. For the following subtasks, it represents the union of all the splits of the outlier data items produced by each worker during the previous subtask ST^{h-1} and sent to worker k:

$$M_k^h = \begin{cases} I_k^0 & \text{if } h = 1 \\ \bigcup_{j \in [K]} O_{j \to k}^{h-1} & \text{if } h > 1 \end{cases}$$

where the set O^{h-1} denotes outliers produced in the previous superstep h_1, and the subscript $j \to k$ denotes items produced by worker j and sent to worker k.

The subtask ST^h produces a set of intermediate results. These intermediate results, together with the output of the execution of the current subtask in the previous superstep, are the input for the outlier detection algorithm

$$I_k^h = ST^h(M_k^h) \bigcup \widehat{ST}^h(R_k^{h-1}).$$

The outlier detection algorithm outlier() separates these items into two groups, according to a specified cost model: O_k^h is the set of outliers, R_k^h is the set of the remaining regular data items.

Outlier detection must guarantee that the load on each worker for its local regular items is balanced. Therefore, the regular items are passed to the next subtask immediately and locally, and produce a new set of intermediate results for the next superstep. We denote this execution of the next subtask as $\widehat{ST}^{h+1}$.

The outlier data items are sent to a split() operator, which takes care of spreading the load of these items across all the workers, and thus create the input for the next superstep M_*^{h+1}. This operator splits the outliers into groups with balanced cost, and sends each group (called a *split*) to a different worker.

Applicability. Fragmentation is suitable for tasks where it is possible to: (*i*) partition the input in data items, such that each of them can be processed in parallel by a task without coordination, (*ii*) split the task into a sequence of subtasks, (*iii*) express intermediate results as a set of data items, (*iv*) provide a cost function for each data item, and finally (*v*) split outliers into a set of data items with similar cost.

The first three points are related to the system model we described before. The cost function mentioned in the fourth point estimates the running time of processing a data item in the next subtask. More precisely, for each subtask ST^h, there must exist a cost function c^h that takes a data item $i \in I_*^{h-1}$ as input and returns an estimate of the time it will take ST^h to process i. Outlier detection uses the cost function to guarantee that the cost of processing the

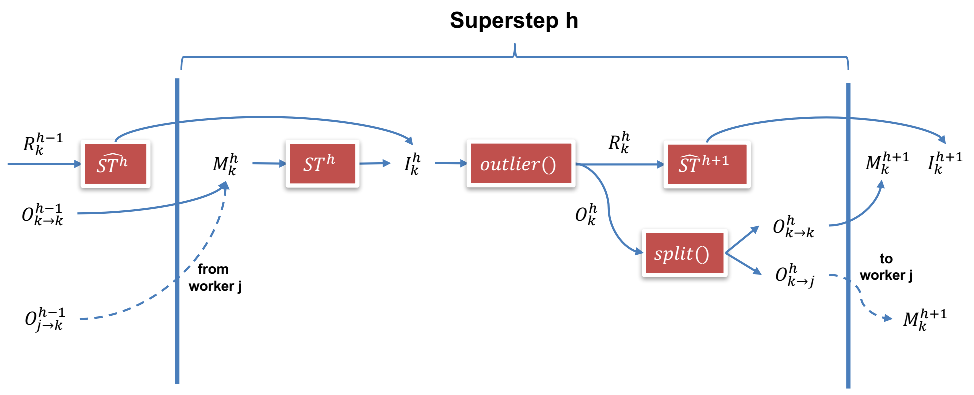

Figure 1: Overview of a single step (subtask) when applying task fragmentation.

regular data items at each worker is balanced, based on some probabilistic assumptions. Finally, the split operator must generate balanced splits for the outlier data items.

In this paper, we show how to use task fragmentation to parallelize graph search, and specifically for tree-based subgraph isomorphism algorithms such as TurboISO [19]. We believe that task fragmentation is interesting in its own regard and can be applied to other problems beyond graph search, however such generalization is beyond the scope of the current paper and left for future work.

3 SUBGRAPH ISOMORPHISM

QFrag is the first distributed system which uses subgraph isomorphism for search. This section gives some background on existing subgraph isomorphism algorithms.

We start by introducing the problem of subgraph isomorphism, and describe the general structure of algorithms that solve this problem, all of which share many similarities.

3.1 Problem Definition

Let $G = (V_G, E_G, L_G)$ be an undirected labeled *data graph*, where V_G is the set of vertices of G, $E_G \subseteq V_G \times V_G$ is the set of undirected edges, and L_G is a labeling function which maps a vertex or an edge to a set of labels. Let $Q = (V_Q, E_Q, L_Q)$ be an undirected labeled *query graph*. Informally, a subgraph isomorphism is a subgraph of the data graph which matches the query graph. In the subgraph isomorphism jargon, the terms subgraph isomorphism and embedding can be used interchangeably. A *partial embedding* is a subgraph of the data graph that at least partially matches the query graph. Each embedding corresponds to a mapping between query vertices and the vertices of the embedding. This mapping is typically defined as an injective function, i.e., a data vertex can map to only one query vertex. However, subgraph isomorphism algorithms can be trivially adapted to relax this constraint.

3.2 Tree-Based Algorithms

We now describe more in details the characteristic of state-of-the-art sequential subgraph isomorphism algorithms such as TurboISO and subsequent work [4, 19, 29]. We call these algorithms *tree-based* because they start by transforming the query graph into a spanning tree and by matching this tree. To clarify the mapping between task fragmentation and subgraph isomorphism, we describe the latter in terms of two phases, tree building and embedding enumeration, which correspond to two subtasks. We now describe these phases more in detail.

Tree building phase. The goal of the first phase is to quickly identify the subgraphs that might match the query. It produces a set of candidate trees, which are used by the subsequent embedding enumeration phase.

The first step in the tree building phase is to identify a root vertex $s_Q \in V_Q$ in the query graph. Different algorithms use different heuristics for this selection. For example, TurboISO favors query vertices that have higher degree in the query graph and whose label has fewer matches in the data graph. Next, the algorithm creates a *spanning query tree* Q_t in the query graph, via a breadth-first exploration from s_Q. The edges of the query graph that are not included in the spanning tree are called *cross-edges*. Figure 2 shows an example of a spanning query tree.

After identifying a spanning tree in Q, the tree building phase builds a candidate tree $CT(r)$ for each vertex $r \in V_G$ of the data graph that matches the root of the spanning query tree s_Q. The root node of a candidate tree $CT(r)$ is the vertex r. The other nodes of the tree are sets of vertices in V_G called *domains*. Each node in the candidate tree, i.e., each domain, corresponds to a node in the spanning query tree. For example, a candidate tree $CT(r)$ for the spanning query tree of Figure 2 has a root r matching q_1, two level-one domains $D(q_2)$ and $D(q_5)$ corresponding to q_2 and q_5 respectively, and three level-two domains.

The tree building phase matches one query vertex at a time, according to a depth-first order on the spanning query tree. It builds a domain by adding neighbors of the vertices in the parent domain. For example, when matching q_3, it adds v to $D(q_3)$ if v has the same label as q_3 and there exists a vertex $v' \in D(q_2)$ such that the edge (v', v) has the same label as the edge (q_2, q_3).

Figure 3 shows how TurboISO stores the candidate tree for the spanning query tree of Figure 2. Each domain $D(q_i)$

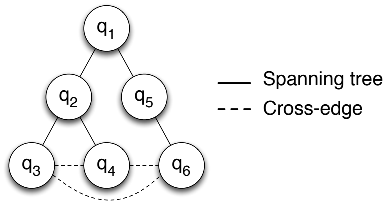

Figure 2: Example of query graph Q. The full edges are part of the spanning query tree Q_t, the dashed edges are cross-edges. The vertex IDs denote the depth-first order in which vertices are matched in the tree building phase.

consists of a set of candidate subregions, where $CS(q_i, v_j)$ is the set of data vertices that match q_i and that can be reached from a vertex v_j in the parent domain.

Embedding enumeration phase. The embedding enumeration phase enumerates partial embeddings based on the candidate tree. For each candidate tree $CT(r)$, it checks whether the cross-edges required by the query graph are present. Partial embeddings are reproduced in a depth-first order, although the order in which the domains of a tree are visited can be changed for efficiency on a tree-by-tree basis. Consider for example the candidate tree of Figure 3, where the query vertices and the domains are explored in the order $\langle q_1, q_2, q_3, q_4, q_5, q_6\rangle$. A possible reordering that still induces a depth-first visit is $\langle q_1, q_5, q_6, q_2, q_4, q_3\rangle$.

The order of query vertices for a given candidate tree $CT(r)$ is called *matching order*. In the example of Figure 3, if the matching order of $CT(r)$ is $\langle q_1, q_5, q_6, q_2, q_4, q_3\rangle$ and the current partial embedding is $M = \langle r, v_3\rangle$ matching $\langle q_1, q_5\rangle$, the next vertices to be added to M are the ones in $CS(q_6, v_3)$. Different algorithms use different reordering heuristics to determine matching orders. Each candidate tree can have a different matching order, based on the cardinality of its domains. We consider TurboISO's heuristic, which orders first the paths having more cross edges and whose leaves have smaller cardinality. These paths have higher selectivity, so they are more likely to reduce the number of embeddings that need to be enumerated.

Cost of the two phases. Tree building is much faster than embedding enumeration, as it does not require rebuilding complete embeddings. Consider, for instance, the candidate tree of Figure 3. In order to populate the domain $D(q_6)$, it is sufficient to scan all data vertices in $D(q_5)$. For each vertex, we add to $D(q_6)$ all the neighbors that match the label of the query vertex q_6 and of the edge (q_5, q_6). Therefore, tree building visits the elements of a domain D only once (or never, if D corresponds to a leaf of the spanning query tree).

Embedding enumeration, instead, entails enumerating all the combinations of vertices (one for each domain) that constitute an embedding. Consider again the candidate tree of Figure 3. Embeddings are enumerated in a depth-first fashion by adding vertices from one domain at a time. The algorithm starts building the partial embedding $\langle r\rangle$, then $\langle r, v_1\rangle$, and so on. After reaching the maximum size of an embedding, which is six in our example, the enumeration backtracks and considers other branches.

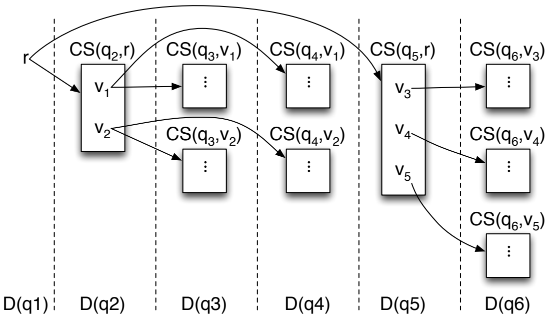

Figure 3: Example of candidate tree for the query graph in Figure 2. The candidate tree represents the candidate tree $CT(r)$ for root data vertex r. Boxes represent candidate subregions. Dashed lines show domains $D(q_i)$, which are sorted in the order in which query vertices are matched during the tree building phase. Edges are parent-child relationships in the spanning query tree.

To see why embedding enumeration is more complex, we look at vertex $v_3 \in D(q_5)$ of Figure 3. As discussed previously, tree building visits v_3 only once. We now count the times v_3 is visited during the embedding enumeration phase. Assume that the matching order is $\langle q_1, q_2, q_3, q_4, q_5, q_6\rangle$. The query graph of Figure 2 has cross edges between q_3, q_4, and q_6. Therefore, we must enumerate all embeddings corresponding to all the combinations of vertices matching q_3, q_4, and q_6, and check the cross edges among these vertices. This implies considering, among others, all combinations of embeddings of the form $\langle r, v_1, v_i, v_j, v_3\rangle$, where $v_i \in CS(q_3, v_1)$ and $v_j \in CS(q_4, v_1)$. Therefore, vertex v_3 in $D(q_5)$ will be visited at least $|CS(q_3, v_1)| \times |CS(q_4, v_1)|$ times. Considering also embeddings of the form $\langle r, v_2, v_i, v_j, v_3\rangle$, we obtain that v_3 is visited $(|CS(q_3, v_1)| \times |CS(q_4, v_1)|) + (|CS(q_3, v_2)| \times |CS(q_4, v_2)|)$ times in total.

4 QFRAG

The QFrag distributed graph search framework aims at scaling out the execution of tree-based sequential subgraph isomorphism algorithms. Its parallelization policy is an instance of the general task fragmentation template described in Section 2. In order to test the efficacy of our proposal, we implement two different parallelization policies for QFrag, *embarrassingly parallel* and *task fragmentation*, which we describe in the following.

4.1 Embarrassingly Parallel

The initial task for QFrag is a sequential tree-based subgraph isomorphism algorithm based on TurboISO, as described in the previous section. The embarrassingly parallel policy assigns the starting data vertex with ID v to the worker $h(v) \bmod p$, where p is the number of workers and h a hash

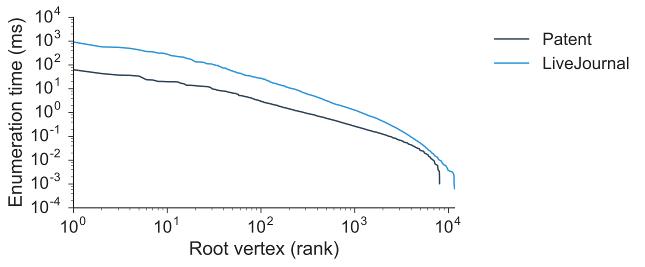

Figure 4: Per-tree distribution of enumeration times for Q5u (unlabeled triangles) on Patent and LiveJournal. The LiveJournal graph has a much more skewed workload.

function. The entire task, comprising both phases of the algorithm (tree building and embedding enumeration) described in Section 3, is run until completion by the assigned worker in one BSP superstep. The embarrassingly parallel nature of this policy stems from the fact that in QFrag each worker has access to a full copy of the input graph, and thus it has all the information to execute the complete subgraph isomorphism algorithm on a candidate tree.

This policy is simple and effective, but has a major limitation. It is well known that many natural graphs are characterized by skew: some vertices are connected to a much larger set of neighbors than others. Indeed, the vertex degrees often follow a power-law distribution [7, 13]. As a result, the size of a candidate region, and thus the amount of work for each root data vertex, can vary significantly.

Figure 4 reports the per-candidate-tree distribution of running time of the embedding enumeration phase for two input graphs: Patent, a citation graph, and LiveJournal, a social network (see Section 5 for more details). The query graph is an unlabeled triangle. The plot, in log-log scale, shows that the range of execution times spans five to six orders of magnitude. In addition, the most expensive trees in the LiveJournal graph take one order of magnitude longer than in the Patent graph. Such heavy skew in the workload hinders effective parallelization of the algorithm.

The embarrassingly parallel policy uniformly partitions root vertices, and thus candidate trees, among the workers in the system. The per-worker skew is lower than the per-tree skew due to aggregation, as shown in Figure 5. For graphs that are not very skewed, such as Patent, the embarrassingly parallel policy is adequate, and task fragmentation performs similarly. However, for skewed graphs such as LiveJournal the difference in load among workers is still significant, so task fragmentation results in a substantial speedup.

4.2 Task Fragmentation

The problem of skew motivates us to use task fragmentation to balance the load in presence of outlier candidate trees, which are expensive candidate trees which carry a disproportionate amount of work.

By using task fragmentation, QFrag is able to share the load of processing heavy trees across the workers. Balancing

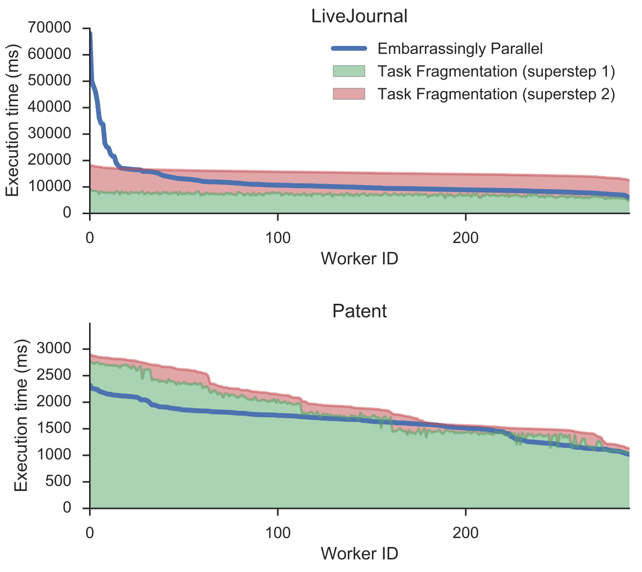

Figure 5: Cumulative execution time per worker with Embarrassingly Parallel and Task Fragmentation parallelization (sum of the two phases) for an example query (Q5u, triangles). For a skewed graph such as LiveJournal, Task Fragmentation creates a much more uniform workload distribution, which is more amenable to parallelization. For a less skewed graph, such as Patent, the gain from Task Fragmentation is not significant enough to compensate the overhead, thus making it slightly more expensive than the Embarrassingly Parallel policy.

load in our setting is particularly challenging because graph exploration with local graph access is extremely fast compared to the cost of coordination. In order to minimize the cost of rebalancing, task fragmentation focuses on identifying the few heavy trees that cause most of the work. The pseudocode for task fragmentation in QFrag is illustrated in Algorithm 1.

Fragmenting to subtasks. The initial task for QFrag is a sequential tree-based subgraph isomorphism algorithm, as described in the previous section. QFrag subdivides the sequential graph exploration logic into two subtasks, *tree building* and *embedding enumeration*, which correspond to the two phases described in Section 3. The TREE BUILDING function (i.e., ST^1 in the notation of Section 2) takes as input one root vertex $r \in V_G$ and outputs a candidate tree $CT(r)$ as intermediate result. A worker can receive multiple root vertices as input and produce multiple candidate trees. The EMBEDDING ENUMERATION function (i.e., ST^2) takes as input one candidate tree $CT(r)$ and outputs the set of embeddings of the query graph that are rooted in r. Following the notation of Section 2, I_k^0 is the set of root vertices received by worker k. Each root vertex represents a data item in I_k^0 and generates a candidate tree, which is a data item in the I_k^1 set. Each final output embedding is a data item in I_k^2.

Algorithm 1: Task Fragmentation for graph search - worker k

```
input  : $I_k^0$: Set of root vertices assigned to worker $k$
input  : $G$: Data graph, $Q$: query graph
output : All subgraphs of $G$ matching $Q$

Superstep 1
foreach $v \in C$ do
    $CT(r) \leftarrow$ TREE BUILDING($v$);
    if $|T| < k$ then
        add $CT(r)$ to $T$;
    else
        $next \leftarrow CT(r)$;
        $CT(r_m) \leftarrow CT(r) \in T$ with minimum cost;
        if $cost(CT(r)) > CT(r_m)$ then
            replace $CT(r_m)$ with $CT(r)$ in $T$;
            $next \leftarrow CT(r_m)$;
        $E_k(r) \leftarrow$ EMBEDDING ENUMERATION($next$);
        output $E_k(r)$;
foreach $CT(r) \in T$ do
    $\{CT_{k \to 1}(r), \ldots, CT_{k \to K}(r)\} \leftarrow$ SPLIT($CT(r)$);
    foreach $i \in [1, K]$ do
        send $CT_{k \to i}(r)$ to worker $i$;

Superstep 2
$M \leftarrow$ set of received candidate trees;
foreach $CT(r) \in S$ do
    $E_k(r) \leftarrow$ EMBEDDING ENUMERATION($CT(r)$);
    output $E_k(r)$;
```

The execution time of TREE BUILDING is orders of magnitude smaller than the one of EMBEDDING ENUMERATION, as we have discussed in Section 3.2. Therefore, determining a good partitioning of the root vertices to give as initial input to TREE BUILDING is not very important in terms of global execution time, because any skew that might arise is not relevant in absolute terms. Focusing on balancing the load of EMBEDDING ENUMERATION is thus sufficient.

QFrag runs the subtasks on a distributed system with multiple workers. It ensures that each worker has a local, read-only copy of the input query and data graphs. Each subtask has access to this input. Many subgraph isomorphism algorithm perform a preprocessing of the query graph, for example in order to build the spanning query tree, or to identify symmetries in the query graph that can simplify enumeration. QFrag executes this preprocessing locally at each worker. The initialization code must be deterministic so that each worker can independently initialize the state without coordination.

Outlier detection. The outlier detection policy for graph search is simple: the outlier candidate trees are the top κ% trees by estimated cost for the next subtask. In the first superstep, each worker performs tree building, and computes the estimated cost of each tree $CT(r)$ it builds by using a cost function that we describe shortly. Each worker also keeps a *priority queue* T of the top κ% most expensive trees it has built so far (outliers). Candidate trees that are not added to T, or that are added and later removed (i.e., regular trees), are given as input to the EMBEDDING ENUMERATION function in the first superstep. The outlier trees are split into multiple *split candidate trees*, one per worker, by a SPLIT function. The goal of the SPLIT function is to produce splits of equal cost. Each split candidate tree is sent to a different worker. Workers process the split candidate trees in the second superstep. The algorithm gives the received splits as input to the EMBEDDING ENUMERATION function, which returns additional embeddings matching the query graph.

After evaluating different values of κ, we found that the value $\kappa = 0.1\%$ works well across different graphs and queries. We conducted sensitivity analysis on a variety of datasets and queries and noted that the query execution time is not very sensitive to moderate variations of this parameter. This is because the execution time in skewed graphs is typically dominated by very few outliers.

We now discuss the function we use to estimate the cost of a tree, and therefore to find outliers, and the algorithm we use to split trees.

Cost estimation. A precise estimation of the cost of the embedding enumeration subtask depends on the specifics of the algorithm at hand. For example, different versions of TurboISO use different optimizations match cross-edges, which may sometimes avoid enumerating some combinations of embeddings [22]. Other algorithms do not immediately enumerate all the partial embeddings in the candidate tree [4]. QFrag does not require a very precise cost estimation, since it only needs to discriminate heavy trees from the rest. Therefore, its cost function is simply a heuristic approximating the number of different embeddings that a given candidate tree can generate.

A candidate tree $CT(r)$ associates a domain to each query vertex except the root r. Let D_r be the singleton domain containing only r. Let D_l be a leaf domain. We estimate the cost of D_l as

$$c(D_l) = |D_l|.$$

The cost of an internal node D is the product of the costs of its children times the size of the domain

$$c(D) = |D| \times \prod_{D' \in \mathrm{Children(D)}} c(D').$$

Finally, we compute the cost of the candidate tree $CT(r)$ as the cost of its root

$$cost(CT(r)) = c(D_r). \tag{1}$$

This metric is not entirely accurate but it is a good approximation in many cases. Outlier detection does not require an accurate cost function: it is sufficient to have a cost function that assigns a relatively higher weight to the trees that require the longest enumeration time. Figure 6 shows the correlation between this cost metric and the execution time of the EMBEDDING ENUMERATION function for each candidate tree of the LiveJournal graph (see Section 5) with an unlabeled triangle query (Q5u). The cost function is able to reliably identify the heaviest candidate trees, even if it tends to be conservative and to underestimate the pruning opportunities present in large trees.

Splitting. The SPLIT function partitions a candidate tree into multiple split candidate trees and makes sure that there is one split per worker. It selects one *split domain* D, partitions

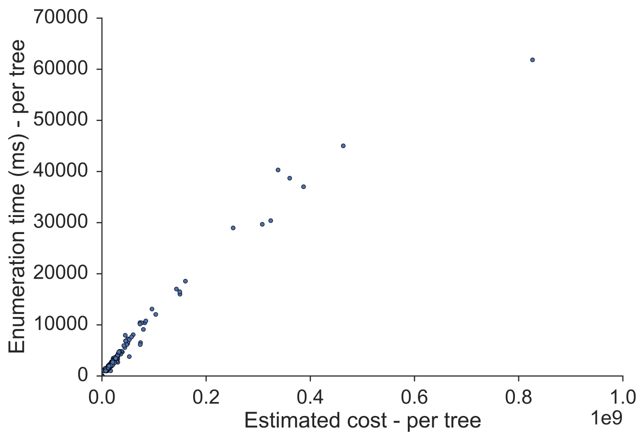

Figure 6: Correlation between estimated cost of a candidate tree (as per Equation 1) and its enumeration time for the LiveJournal graph with query Q5u (unlabeled triangles). Each point represents one candidate tree. The cost function of Equation 1 is able to reliably identify the most expensive trees.

the vertices in D and in its children domains among the splits, and copies all other domains that are not children of D. In other words, QFrag splits the Cartesian product over the domain of a single query vertex.

Given that the EMBEDDING ENUMERATION function enumerates the Cartesian product of the domains in independent branches of the candidate tree, splitting along any sufficiently large domain (i.e., with at least one vertex per worker) yields similarly sized splits. Thus, QFrag selects a level-one domain (i.e., a child of the root domain) as split domain. More specifically, QFrag selects the first domain in the matching order. This is the first domain (after the root domain) to be visited in the embedding enumeration phase, so splitting it in equal parts gives the highest likelihood of splitting the load in a balanced manner.

In order to split a domain $D(u_q)$, the system partitions its elements across the newly created splits of the candidate tree, and we say that u_q is split. For instance, let a vertex $v'_G \in CS(u_q, v_G)$ be assigned to a certain split. All children candidate subregions $CS(u'_q, *)$ reachable through v'_G are also assigned to the same candidate tree split. At the end of the splitting process, all other candidate subregions for query vertices that have not been split are copied over to every candidate tree split.

Referring to the example of Figure 3, QFrag produces two splits s_1 and s_2, each with half of the candidate subregion $CS(q_2, r)$. Therefore, s_1 includes $CS(q_2, r) = \{v_1\}$ and s_2 includes $CS(q_2, r) = \{v_2\}$. Then, QFrag partitions candidate subregions that are reachable from the split candidate subregion, so s_1 also includes $CS(q_3, v_1)$ and $CS(q_4, v_1)$, which are reachable from v_1, whereas s_2 includes $CS(q_3, v_2)$ and $CS(q_4, v_2)$. Finally, all remaining candidate subregions are copied to both splits s_1 and s_2.

This algorithm for splitting candidate trees has two correctness properties. First, the splitting algorithm does not generate redundant embeddings: the embeddings in different splits do not overlap. To see why this property holds, consider an embedding $e = \langle v_1, \ldots, v_n \rangle$ that is generated from the original tree before the split and let i be the index of the split domain. By construction, vertex v_i of e can only be assigned to a single split since its domain is partitioned, so e cannot be generated from two different splits.

The second correctness property of the splitting algorithm is that no embedding is lost. Consider again the embedding e discussed previously. We know that its vertex v_i is assigned to a single split s. Consider now another vertex v_j in e with $j \neq i$. Let CS_i and CS_j be the candidate subregions containing vertex v_i and v_j, respectively. If CS_j is reachable from CS_i, then by construction CS_j is assigned to the same split s as v_i. If CS_j is *not* reachable from CS_i, then CS_j is will be replicated on all splits, again by construction. In both cases, an embedding containing both vertices v_i and v_j is generated by running embedding enumeration on s.

Implementation. We implemented QFrag in Java on top of Hadoop YARN and Apache Giraph. Our implementation does not follow a "think like a vertex" approach: it uses Giraph as a BSP execution engine. The two steps of Algorithm 1 correspond to synchronous supersteps according to the Bulk Synchronous Parallel (BSP) model. QFrag's inputs are the data and query graphs, which are provided as HDFS paths. Its output is the set of matches, which can be either printed or written to HDFS. We plan to implement QFrag also as a Spark library and to release it as open source.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of QFrag versus alternative computation paradigms. Our aim is to discover the limits of the systems and understand how they behave under significant workloads.

We focus on the following research questions:

RQ1: What is the performance of QFrag compared to the baselines in sequential execution;

RQ2: What is the performance of QFrag compared to the baselines in distributed execution;

RQ3: What is the impact of using task fragmentation over an embarrassingly parallel implementation.

5.1 Experimental Setup

Datasets. The input graphs are medium to large scale, as shown in Table 1. Patent [17] contains citation edges between US Patents between January 1963 and December 1999; the label of a patent is the year when it was granted. YouTube [8] lists crawled videos and their related videos posted from February 2007 to July 2008. The label is a combination of a video's rating and length. Orkut and LiveJournal are social networks from the KONECT archive, and are unlabeled.[3]

[3]http://konect.uni-koblenz.de/networks/orkut-links, http://konect.uni-koblenz.de/networks/soc-LiveJournal1

Table 1: Statistics for the graphs used for the evaluation: number of vertices $|V|$, edges $|E|$, and labels $|L|$; average μ_d, standard deviation σ_d, and coefficient of variation $CV_d = {}^{\sigma_d}/_{\mu_d}$ of the degree distribution.

Name	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert L\rvert$	μ_d	σ_d	CV_d
Patent	2.7mln	14mln	37	5.09	13.94	2.74
YouTube	4.6mln	44mln	108	9.58	27.30	2.85
Orkut	3.1mln	117mln	0	38.14	168.29	4.41
LiveJournal	4.8mln	43mln	0	8.84	54.22	6.13

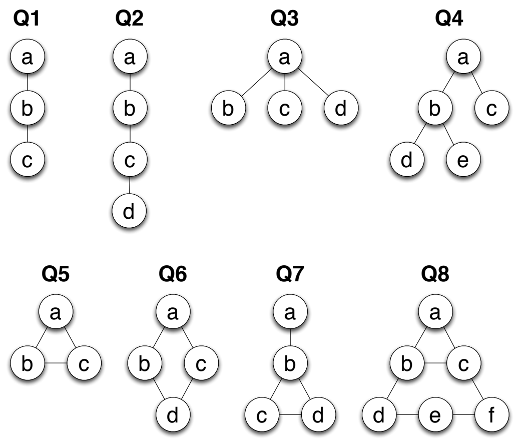

Figure 7: Queries used in the evaluation. The queries represent four different topologies: chains (Q1-Q2), trees (Q3-Q4), cycles (Q5-Q6), and mixed (Q7-Q8). We consider two variants of each query. In the labeled version, the label of each vertex is mapped, in alphabetical order, to the most frequent labels of each graph in decreasing order of frequency, i.e., 'a' represent the most frequent label in the graph, 'b' the second most frequent, and so on. In the unlabeled (structural) version, the query vertices are unlabeled so any data vertex can match every query vertex.

These two datasets have much higher skew in the degree distribution (see the coefficient of variation CV in Table 1), and are therefore more challenging for load balancing.

Queries. We generate queries for these datasets in a programmatic fashion. We start with 8 template query topologies, as shown in Figure 7. These queries are similar to ones already used in the literature [23]. These topologies present a variety of chains, trees, cycles, and mixed ones, and are mostly taken from the literature [2]. Some queries, such as Q5 and Q6, have direct applications to network evolution models and recommender systems [15, 24]. We consider two variants for each query. In the labeled variant, we instantiate these queries for each dataset by applying a label to each vertex in the template. It assigns label 'a' in the query template to the most frequent label in the given graph, label 'b' to the second most frequent, and so on. In the unlabeled variant, we use these queries as structural queries: the query vertices can match any data vertex. We append the letter u to the name of the queries in their unlabeled version, as in Q5u.

Two of our input datasets are labeled, while the other two are unlabeled. We run both labeled and unlabeled queries on the labeled graphs (Patent and YouTube), and only unlabeled queries on the unlabeled graphs (Orkut and LiveJournal).

Environment. We run our experiments on a cluster of 10 servers. Each server has 2 Intel Xeon E5-2670@2.67GHz CPUs with a total of 32 execution cores and 256GB of RAM. The servers are connected with a dual 10 GbE network. We configure Hadoop 2.6.0 so that each physical server contains a single worker with 32 execution slots.

Baselines. We use four different systems as baselines:

Neo4j is a popular production-grade graph database.[4] The database is centralized and spawns a single thread per query. We use Neo4j community version 2.3.2.

VF2 is a well-known algorithm for sub-graph isomorphism [10]. We use the implementation available in the C++ Boost library.[5] We did not use TurboISO as baseline because its source code is not available.

TriAD is a state-of-the-art distributed shared-nothing RDF engine implemented in C++ and MPI [16]. The system is based on an asynchronous distributed join algorithm which uses a partitioned locality-based index.

GraphFrames (GF) is the Apache Spark[6] package that extends Spark's functionality to handle graph datasets as native Spark DataFrames. GraphFrames[7] provides a high-level API for querying the graph, and transform the queries in an optimized SparkSQL execution plan. Therefore, it is based on executing distributed joins on top of Spark. We use the latest available version of GraphFrames available on GitHub as of May 2016[8] and run it on the latest stable version of Spark (1.6.1).

For each system, we report the query response time, excluding loading the graph and any initial pre-processing and indexing, and including the output phase.

5.2 Sequential Efficiency (RQ1)

In this first experiment we compare QFrag with the other systems running sequentially. The goal of this experiment is to quantify the overhead that might be present in QFrag, compared to other solutions, including ones that are sequential (e.g., VF2) or other distributed solutions specifically designed for graph search (e.g., TriAD). For this evaluation we use labeled queries, because they can be executed by all systems in sequential mode.

[4] http://neo4j.com
[5] http://www.boost.org/doc/libs/master/libs/graph/doc/vf2_sub_graph_iso.html
[6] http://spark.apache.org
[7] http://graphframes.github.io
[8] http://github.com/graphframes/graphframes/tree/8a7f973422f0302496a0dfbc0dabbdc2db6af338

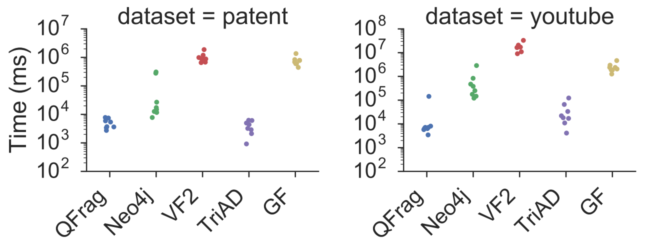

Figure 8: Response time with labeled queries and *sequential* execution, in log scale. Each dot in the graph represents one of the eight test queries in Figure 7.

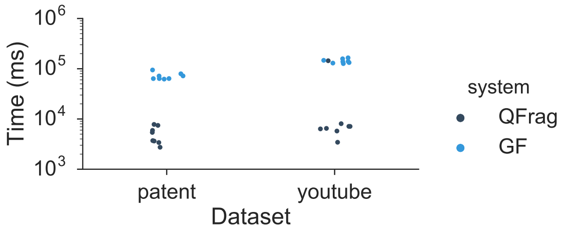

Figure 9: Response time of GraphFrames (320 processes) vs. QFrag (1 process) using labeled queries. Each dot represents one test query.

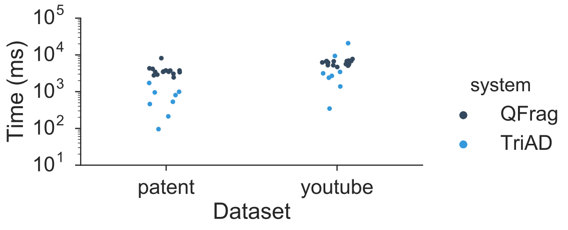

Figure 10: Response time of TriAD (32 processes) vs. QFrag (32 workers) using labeled queries. Each dot represents one test query.

Figure 8 reports the running times of all 8 labeled test queries for the labeled datasets on the systems running in sequential mode. We plot the graph in log scale because different systems have very different running times. Overall, QFrag is one of the two best systems, together with TriAD, and outperforms the other competitors by orders of magnitude. QFrag outperforms TriAD for more complex, long running queries, where gains are more significant in absolute terms. For example, with Q8 QFrag is 3× and 16× faster than TriAD on Patent and YouTube, respectively. VF2 does not scale to larger graphs gracefully, and indeed, it is the slowest system on YouTube. For complex queries such as Q8, QFrag is 327× and 2088× faster on Patent and YouTube, respectively. Neo4j offers reliable performance, even though the response times are not lightning fast. Possibly, the focus on non-functional aspects such as durability and persistence, and the fact that it is a disk based system, put a lower bound on the response times. For all queries QFrag is typically at least one order of magnitude faster than Neo4j. GraphFrames

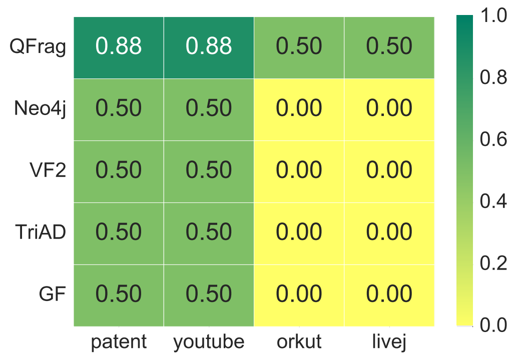

Figure 11: Fraction of queries that each system is able to successfully run for each dataset (in any configuration). All systems are able to run labeled queries, and while no other baseline is able to run any unlabeled query, QFrag successfully runs half of them. Note that Orkut and LiveJournal are unlabeled graphs so we consider only unlabeled queries, whereas in Patent and Youtube only half of the queries we consider are unlabeled.

(GF) is one of the slowest system. While it scales pretty well, as shown next, its additional overhead is too high to compete with other systems. QFrag is regularly faster than GF by more than two orders of magnitude.

Overall, these results show that QFrag does not sacrifice efficiency for scalability, rather it is competitive with centralized and highly optimized implementations. This is not surprising given that QFrag implements TurboISO, a state of the art sequential subgraph isomorphism algorithm.

5.3 Distributed Efficiency (RQ2)

The purpose of this second experiment is to compare QFrag to other distributed solutions, namely, TriAD and GraphFrames. Both these systems partition the input graph across the worker machines, and rely on distributed joins to answer graph search queries. QFrag takes a different approach: it replicates the input graph on the workers, and uses graph exploration to distribute the work. In other words, QFrag distributes the computation, not the data.

Labeled queries. We first compare QFrag with GraphFrames, which is the system that is most similar to QFrag in terms of technology. It is also an in-memory graph search system built on top of Hadoop, and like QFrag, it lets workers coordinate via synchronized communication steps. We compare GraphFrames running with a parallelism of 320 (the maximum possible on our cluster) to the sequential version of QFrag (Figure 9). GraphFrames shows good scalability compared to its running time in the sequential mode, reported in Figure 8. The improvement is typically around one order of magnitude. However, this is still not sufficient to outperform the sequential execution of QFrag, which is still more than one order of magnitude faster across all queries. Therefore, GraphFrames shows a high COST (configuration

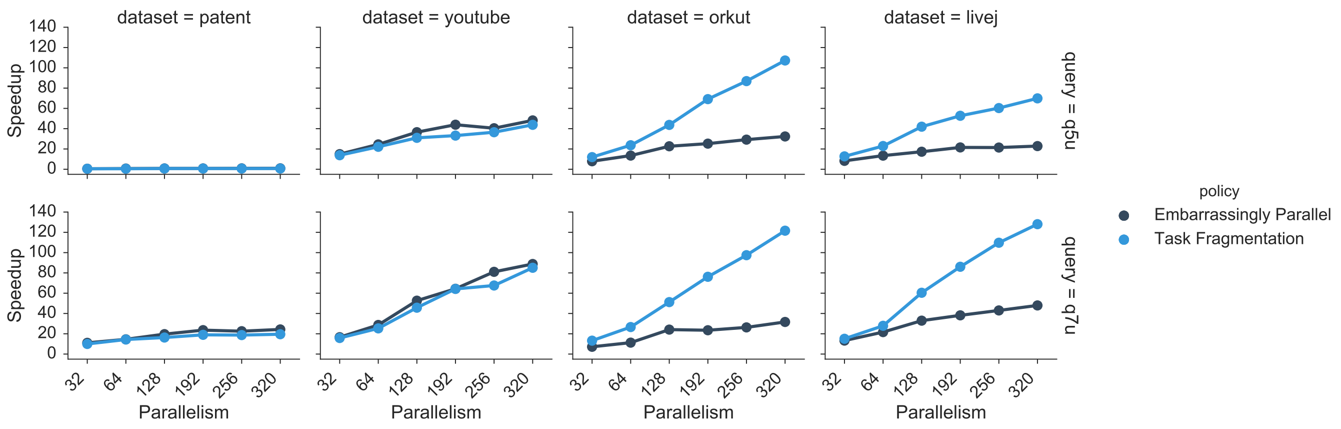

Figure 12: Speedup of different distribution policies for QFrag over sequential execution. For datasets with low skew (Patent and YouTube), embarrassingly parallel is as good as task fragmentation. However, for datasets with high skew (Orkut and LiveJournal), task fragmentation is significantly better than embarrassingly parallel.

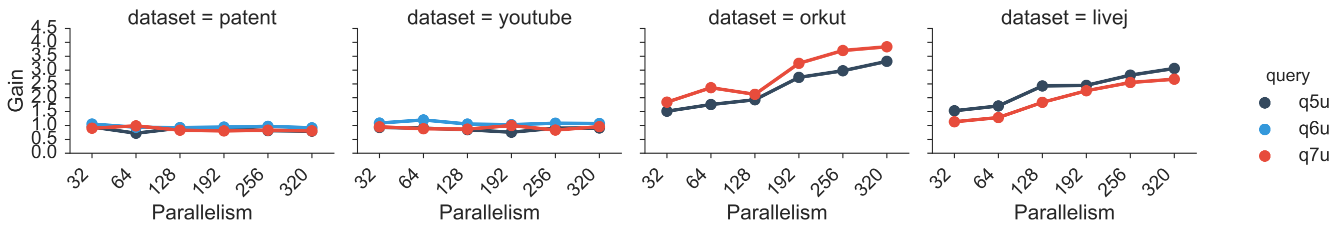

Figure 13: Speedup of task fragmentation over embarrassingly parallel for the same configuration of dataset, query, and processes. For datasets with low skew (Patent and YouTube) there is no difference, while for datasets with high skew (Orkut and LiveJournal) task fragmentation is up to 4 times faster than embarrassingly parallel. Note that QFrag can process query Q6u for Orkut and LiveJournal only with task fragmentation, so it is impossible to compute gains.

to outperform single thread) [27]. Conversely, QFrag does not present this problem.

Next, we compare QFrag to TriAD. The two systems use very different technology stacks: TriAD is implemented in C++ and runs on top of a tailored, asynchronous MPI platform, whereas QFrag is written in Java and runs on top of a Hadoop-based BSP system. Coordination in a BSP system is much more expensive than in MPI because every message exchange requires blocking all workers and waiting until all workers have finished receiving their messages. The purpose of this experiment is to show that, despite these additional constraints, QFrag has performance in line with TriAD, which in turn has been shown to clearly outperform other Hadoop-based solutions [16]. TriAD is not designed to scale to more than 32 processes, so we run both TriAD and QFrag with 32 processes to compare their performance. The results, reported in Figure 10, show a trend similar to the comparison with sequential execution. TriAD outperforms QFrag on the smaller dataset and simpler queries. The running times for QFrag are all very similar, due to a constant overhead of running on top of Hadoop. In relative terms, TriAD can be more than ten times faster, but in absolute terms, the differences are in the order of few seconds, which is not a very significant difference even in case of interactive analytics. For more complex queries with a larger number of query vertices and larger intermediate results, such as Q8, QFrag is faster than TriAD. This difference is particularly evident on larger datasets such as YouTube, where QFrag is more than five times faster than TriAD.

QFrag compares favorably to GraphFrames and TriAD because the latter distribute the input graph. They can store larger graphs, but they must also shuffle a larger fraction of an already large intermediate state across workers.

Unlabeled (structural) queries. Surprisingly, all our baselines are unable to run any of the unlabeled queries on our datasets, and crash. These results are summarized in Figure 11, which shows the fraction of queries each system is able to run on different datasets. Most systems are able to run unlabeled queries on smaller test datasets, but as the scale of the task grows all the baselines fail.

QFrag is able to run most queries on most datasets. However, Q8u presents a challenge even for our system. The system does not crash, but it takes more than 12 hours to complete, so we abort it. This result is a reminder that even though QFrag is a substantial improvement over the state-of-the-art, the task is still NP-hard and there will always be limits to the complexity of queries we can hope to run. These unlabeled queries have a high degree of symmetry, so they could benefit from techniques for exploiting symmetry [19, 29]. Extending QFrag to these techniques is left as future work.

Overall, these results show that QFrag consistently outperforms other systems in processing more complex queries on larger dataset.

5.4 Parallelization Policies (RQ3)

The QFrag framework allows to transparently scale a centralized graph exploration implementation to several servers, by using different underlying work distribution policies. We now compare the two policies, embarrassingly parallel and task fragmentation, using from 32 to 320 workers. We report results on all datasets for three unlabeled queries that can run on all the range of workers: Q5u, Q6u and Q7u. As discussed previously, only QFrag is able to run these queries on the datasets we consider. In order to isolate the overhead of writing to HFDS, in these experiments QFrag does not output the embeddings.

Figure 12 reports the speedup of the different variants of QFrag over the sequential execution. The speedup when using embarrassingly parallel with 128 workers is 4 times higher than the one using 32 workers. When using more workers, the gains are still present but less significant. The scalability gain with embarrassingly parallel distribution is in 25-65% range for Patent, and 34-83% for YouTube. This decrease in efficiency is due to the presence of skew in the workload, which hinders further parallelization.

The high efficiency of local computation in QFrag makes the cost of coordination comparatively higher, especially at larger scale, so the relative gains of increasing scalability become lower as the size of the cluster increases. As observed also by McSherry et al. [27], scalability is easier to achieve in systems with worse baseline performance because there is more margin to gain. For example, GraphFrames scales much better than QFrag but does not manage to achieve the same overall performance.

The task fragmentation policy is able to mitigate skew significantly, as already shown in Figure 5. The benefits of using task fragmentation depend on the graph and query under consideration. In fact, sharing load also entails costs in terms of message serialization and of coordination with other workers. On the two low-skew graphs (Patent and YouTube), task fragmentation does not present significant gains over embarrassingly parallel, and its overhead makes the system slightly slower. Conversely, on the high-skew graphs (Orkut and LiveJournal), task fragmentation shows a much better performance over embarrassingly parallel.

To visualize the difference between the two policies more clearly, Figure 13 shows the speedup of task fragmentation over embarrassingly parallel for each configuration. While for Patent and YouTube there is basically no speedup, task fragmentation is up to 4× faster than embarrassingly parallel on Orkut and YouTube. The difference becomes more significant with higher parallelism, which indicates that task fragmentation scales better than embarrassingly parallel, as expected. The architecture of QFrag, which distributes load and not data, results in very efficient local graph exploration even with larger graphs and with analytical queries.

Task fragmentation shows consistent better results overall, with low overhead when the workload is not skewed, and high gains when it is.

6 RELATED WORK

There is has been a large volume of work on distributed graph search. QFrag differs from existing work because of three key design choices: (i) running on BSP systems, (ii) replicating the input graph at each worker, and (iii) running independent parallel instances of a sequential graph matching algorithm (with load balancing). Previous work looked at different points in the design space, as we now discuss.

RDF search systems. The Resource Description Framework (RDF) format is often used for web metadata and for general knowledge management. It builds a graph among entities (i.e., vertices) by expressing edges as subject-predicate-object triplets. Several RDF databases allow storing and querying RDF graphs. We restrict our discussion to state-of-the-art distributed RDF systems.

TriAD partitions the graph into multiple machines, keeping copies of each tuple for performance. It also uses graph summarization to avoid looking at parts of the graph that cannot contain results. Query plans combine three operations: Distributed Index Scan, Distributed Merge Join, and Distributed Hash Join. Our evaluation shows that QFrag is slower than TriAD in queries that take milliseconds because it has a constant latency of 2-3 seconds, which is acceptable for analytic workloads. Such a small gap is remarkable since TriAD is implemented in C++ using efficient asynchronous MPI communication, whereas QFrag is implemented in Java and runs on top of BSP systems, so it must communicate using expensive synchronous supersteps with global barriers. The advantage of using QFrag increases with the complexity of the query, making QFrag the only system able to deal with very complex queries. These queries make the advantage of distributing computation, instead of data, more evident.

AdPart is a more recent RDF search system that, like TriAD, distributes the input graph among several servers [20]. It is implemented using C++ and MPI. Unlike TriAD, which uses a static partitioning of the input graph, it incrementally redistributes the input graph based on access frequency. The system targets queries whose running time ranges from a few seconds to milliseconds, and shows that it can improve over TriAD for these queries.

Dream [18] is an RDF search system that replicates the graph at all servers. Contrary to QFrag, Dream partitions the query graph into a number of sub-graphs, each of which is handled by a separate server. Dream then performs a form of distributed joins of the intermediate results. The goal of query partitioning is to find the best number of servers to execute a query. The maximum number of servers Dream can use for a given query is bounded by the number of the query sub-graphs, which in turn is bounded by the size of the query graph. Dream also reduces the amount of information that must be exchanged to execute distributed joins: since each server has a full copy of the database, servers can simply exchange metadata instead of actual data.

Trinity.RDF [1] is a distributed graph search system that utilizes an independent key-value store to access the graph. It represents the graph as a sequence of adjacency lists, one per vertex. Each adjacency list is stored as a key-value pair in the Trinity store, where the key is the vertex id and the value is the adjacency list [35]. Therefore Trinity.RDF uses the standard approach of distributing the graph across multiple machines. Trinity.RDF uses a centralized query proxy to compute a query plan based on graph statistics or information from indexes. The query plan can either expand a sub-query or combine two sub-queries and thus can generate disjoint exploration sub-queries. The exploration is edge based. The design necessitates a centralized last phase to join all the sub-queries, and cross-edges, thus while their solution has a smaller intermediate state compared to distributed joins, it still is not as optimized as pure subgraph isomorphism. The purpose of query planning in Trinity.RDF is to minimize communication cost rather than the computational cost. TriAD, the baseline that we use in our evaluation, has been shown to typically outperform Trinity.RDF [16].

Graph search on BSP-based systems. Systems such as H-RDF-3X [21] and SHARD [31] are built on top of Hadoop, like QFrag. They partition the data on HDFS and use MapReduce to coordinate among workers. Query processing is performed using joins, each corresponding to one map-reduce iteration. Unlike QFrag, these systems are disk-based, and have been shown to have substantially inferior performance compared to TriAD [16], which is one of the baselines we use in the evaluation.

Other alternatives to QFrag in the BSP world are algorithms for distributed subgraph listing, which find *unlabeled* patterns using joins and distribute the data graph among multiple servers [23, 36]. QFrag supports also labeled queries, beyond unlabeled ones.

Load balancing. Because of their rigid computational model, BSP systems pose unique load balancing challenges compared to other types of data processing systems, such as for example stream processing systems [6, 9, 30, 33, 38, 39]. Prior work on load balancing and straggler mitigation in BSP systems considers tasks as pre-defined black-boxes whose functionality cannot be modified, and focuses on task scheduling techniques [3, 11, 42].

Task fragmentation also addresses the straggler problem but it takes a different approach: it breaks tasks into multiple sub-tasks and balances load by shuffling intermediate data between these sub-tasks.

A well-known scheduling technique to balance load among a set of workers, especially in multi-threaded settings, is work stealing (see for example [5]). In a distributed setting, work stealing requires workers to pull work from others whenever they are idle. This approach is difficult to implement on top of BSP systems. Every time a fast worker wants to pull some work, it needs to block and wait for the end of the current superstep. Slower workers need to proactively interrupt their work before they can be even contacted by the fast worker, terminate the superstep, and check if there is need to redistribute the work. This requires additional supersteps, may frequently and unnecessarily interrupt computation for all workers, and can result into the same sort of load balancing problems we are trying to solve. In addition, BSP systems are designed for push-style communication, where senders proactively stop computation and send data to others. Executing a pull in a BSP system requires two supersteps: in the first superstep, a worker contacts other workers from which it wants to pull; in the second superstep it receives messages from the contacted workers.

Task fragmentation uses a push-based approach where workers execute a balanced amount of work and push the rest to the other workers. This choice makes it a technique that can be easily used on top of any BSP system.

7 CONCLUSION

Graph search is a well-studied problem in the literature. Existing work on systems for distributed graph search has mainly focused on serving queries with high selectivity on large graphs. However, the graph search problem is NP-hard, so running analytical queries even on small graphs can quickly become computationally intensive. For this reason, there exists a rich literature on sequential algorithms for subgraph isomorphism that are designed to optimize the graph exploration, which is the real bottleneck. In addition, a large majority of graph datasets can be represented in an efficient binary form that can fit in the memory of a single machine, especially given the increased availability of cheap and large main memory.

Based on these observations, we have proposed QFrag, a system for distributed graph search that is based on two fundamental design choices: replicating the input graph at every worker, and parallelizing efficient subgraph isomorphism algorithms. QFrag is able to run complex analytical queries that no other system can run. To do so, QFrag employs *task fragmentation*, a load balancing technique designed to deal with deceitfully parallel problems and stragglers in BSP systems. We show that task fragmentation improves over a naïve strategy by up to four times.

Overall, we believe that the design principles introduced by QFrag open up many interesting research questions in terms of how to optimize load balancing in BSP systems, and QFrag represents just a first step in this direction.

ACKNOWLEDGEMENTS

The authors would like to thank Rade Stanojevic for his help with previous drafts of this paper.

REFERENCES

[1] 2013. *A Distributed Graph Engine for Web Scale RDF Data.* http://research.microsoft.com/apps/pubs/default.aspx?id=183717

[2] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *IEEE International Conference on Data Engineering (ICDE).*

[3] Ganesh Ananthanarayanan, Srikanth Kandula, Albert G Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in Map-Reduce clusters using Mantri.. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lin Quin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[5] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[6] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM SIGMOD international conference on Management of data (SIGMOD).*

[7] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys (CSUR)* 38, 1 (2006), 2.

[8] Xu Cheng, Cameron Dale, and Jiangchuan Liu. 2008. Dataset for "Statistics and social network of YouTube videos". http://netsg.cs.sfu.ca/youtubedata/. (2008).

[9] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. 2003. Scalable distributed stream processing. In *CIDR.*

[10] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008).

[12] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer system. In *CIDR.*

[13] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, Vol. 29. 251–262.

[14] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics* 3, 2-3 (Oct. 2005), 158–182.

[15] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[16] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[17] Hall B. H., A. B. Jaffe, and M. Trajtenberg. 2001. The NBER patent citation data file: Lessons, insights and methodological tools. http://www.nber.org/patents/. (2001).

[18] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. 2015. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *Proceedings of the VLDB Endowment* 8, 6 (2015), 654–665.

[19] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. TurboISO: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD).* 337–348.

[20] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *The VLDB Journal* 25, 3 (2016), 355–380.

[21] Jiewen Huang, Daniel J Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1123–1134.

[22] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *Proceedings of the VLDB Endowment* 8, 11 (2015).

[23] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 10, 3 (2016), 217–228.

[24] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. 2008. Microscopic evolution of social networks. In *ACM SIGKDD international conference on Knowledge discovery and data mining (KDD).*

[25] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudre-Mauroux. 2017. Dependency-driven analytics: A compass for uncharted data oceans. In *Conference on Innovative Data Systems Research (CIDR).*

[26] Brian McBride. 2002. Jena: A semantic web toolkit. *IEEE Internet computing* 6, 6 (2002), 55.

[27] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS).*

[28] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (2010), 91–113.

[29] Xuguan Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

[30] Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. Efficient key grouping for near-optimal load balancing in stream processing systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems.* ACM, 80–91.

[31] Kurt Rohloff and Richard E Schantz. 2011. Clause-iteration with MapReduce to scalably query datagraphs in the SHARD graph-store. In *International workshop on*

Data-intensive distributed computing.

[32] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. 2012. Nobody ever got fired for using Hadoop on a cluster. In *International Workshop on Hot Topics in Cloud Data Processing.*

[33] Mehul A Shah, Joseph M Hellerstein, Sirish Chandrasekaran, and Michael J Franklin. 2003. Flux: An adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on.* IEEE, 25–36.

[34] Ron Shamir and Dekel Tsur. 1997. Faster subtree isomorphism. In *Israeli Symposium on Theory of Computing and Systems.*

[35] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[36] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *ACM SIGMOD International Conference on Management of Data (SIGMOD).*

[37] Zhao Sun, Hongzhi Wang, Bin Shao, Haixun Wang, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proceedings of the VLDB Endowment* (2012).

[38] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, David Garcia-Soriano, Nicolas Kourtellis, and Marco Serafini. 2015. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *International Conference on Data Engineering (IDCE).*

[39] Muhammad Anis Uddin Nasir, Gianmarco De Francisci Morales, Nicolas Kourtellis, and Marco Serafini. 2016. When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing. In *International Conference on Data Engineering (ICDE).*

[40] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990).

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud).*

[42] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments.. In *USENIX Symposium on Operating Systems Design and Implementation.*