Scalable SPARQL Querying of Large RDF Graphs

Jiewen Huang
Yale University
huang-jiewen@cs.yale.edu

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

Kun Ren
Northwestern Polytechnical University, China
renkun_nwpu@mail.nwpu.edu.cn

ABSTRACT

The generation of RDF data has accelerated to the point where many data sets need to be partitioned across multiple machines in order to achieve reasonable performance when querying the data. Although tremendous progress has been made in the Semantic Web community for achieving high performance data management on a single node, current solutions that allow the data to be partitioned across multiple machines are highly inefficient. In this paper, we introduce a scalable RDF data management system that is up to three orders of magnitude more efficient than popular multi-node RDF data management systems. In so doing, we introduce techniques for (1) leveraging state-of-the-art single node RDF-store technology (2) partitioning the data across nodes in a manner that helps accelerate query processing through locality optimizations and (3) decomposing SPARQL queries into high performance fragments that take advantage of how data is partitioned in a cluster.

1. INTRODUCTION

The proliferation of RDF (Resource Description Framework [2]) data is accelerating. The reasons for this include: major search engines such as Google (RichSnippets) and Yahoo (SearchMonkey) displaying Webpages marked up with RDFa more prominently in search results (thereby encouraging increased used of RDFa), popular content management systems such as Drupal offering increased support for RDF, and more data sets becoming available to the public (especially from government sources) in RDF format. Although the increased amount of RDF available is good for Semantic Web initiatives, it is also causing performance bottlenecks in currently available RDF data management systems that store the data and provide access to it via query interfaces such as SPARQL.

There has been significant progress made around the research effort of building high performance RDF data management systems that run on a single machine. This started with early work on Sesame [9], Jena [33], 3store [18], and RDFSuite [7], and continued through many new systems that have been proposed in the last few years [12, 4, 24, 32, 29, 25, 15]. These systems have demonstrated great performance on a single machine for data sets containing millions, and, in some cases, billions of triples. Unfortunately, as the amount of RDF data continues to scale, it is no longer feasible to store entire data sets on a single machine and still be able to access the data at reasonable performance. Consequently, the requirement for clustered RDF database systems is becoming increasingly important.

There has been far fewer research progress made towards clustered RDF database systems. Those that are currently available, such as SHARD [27], YARS2 [19], and Virtuoso [26] generally hash partition triples across multiple machines, and parallelize access to these machines as much as possible at query time. This technique has proven to work well for simple index lookup queries, but for more involved queries, such as those found in popular RDF benchmarks, efficiency is far from optimal. This is because these systems have to ship data around the network at query time in order to match complex SPARQL patterns across the RDF graph. These (potentially multiple) rounds of communication over the network can quickly become a performance bottleneck, leading to high query latencies. Furthermore, these systems use storage layers that were not originally designed for RDF data, and are therefore less efficient on a single node than the state-of-the-art RDF storage technology cited above.

In this paper, we describe the design of a horizontally scalable RDF database system that overcomes these limitations. We install a best-of-breed RDF-store on a cluster of machines (in this paper we use RDF-3X [24] since we found this to be the fastest single-node RDF-store in our internal benchmarking) and partition an RDF data set across these data stores. Instead of randomly assigning triples to partitions using hash partitioning, we take advantage of the fact that RDF uses a graph data model, so we use a graph partitioning algorithm. This enables triples that are close to each other in the RDF graph to be stored on the same machine. This results in a smaller amount of network communication at query time, since SPARQL queries generally take the form of graph pattern matching [3] and entire subgraphs can be matched in parallel across the high performance single-node RDF stores.

In order to maximize the percentage of query processing that can be done in parallel, we allow some overlap of data across partitions, and we introduce a method for automatic decomposition of queries into chunks that can be performed independently, with zero communication across

Proceedings of the VLDB Endowment, Vol. 4, No. 11

partitions. These chunks are then reconstructed using the Hadoop MapReduce framework. We leverage recent work on HadoopDB [5] to handle the splitting of query execution across high performance single-node database systems and the Hadoop data processing framework.

After presenting the architecture of our system (Section 3) along with our algorithms for data and query partitioning (Sections 4 and 5), we experimentally evaluate our system against several alternative methods for storing and querying RDF data, including single-node database systems and scalable clustered systems. Overall, we find that our techniques are able to achieve up to 1000 times shorter query latencies relative to alternative solutions on the well-known LUBM RDF benchmark.

In summary the primary contributions of our work are the following:

- We introduce the architecture of a scalable RDF database system that leverages best-of-breed single node RDF-stores and parallelizes execution across them using the Hadoop framework.
- We describe date partitioning and placement techniques that can dramatically reduce the amount of network communication at query time.
- We provide an algorithm for automatically decomposing queries into parallelizable chunks.
- We experimentally evaluate our system to understand the effect of various system parameters and compare against other currently available RDF stores.

Although this paper focuses on RDF data and SPARQL queries, we believe that many of our techniques are applicable to general graph data management applications, especially those for which subgraph matching is an important task.

Furthermore, while our system is primarily designed for the largest RDF data sets, we will also show that even for smaller data sets, the techniques introduced in this paper are highly relevant. This is because some of the more complicated tasks that people want to do with RDF (such as inference) perform orders of magnitude faster if the entire working dataset can be held in memory. A dataset that would normally be disk-resident can be partitioned into the main memory of a cluster of machines using the horizontally scalable architecture described in this paper. Consequently, for some of these tasks, we observe super linear speed-up in our experiments, as more nodes are added to a cluster and disk-resident data are stored entirely in distributed memory.

2. BACKGROUND

2.1 RDF

The "Resource Description Framework," [2] or RDF, is a data model that was proposed by the W3C as a standard for representing metadata about Web resources, and has become a popular data model for releasing public data sets on the Internet. Its schema-free model makes RDF a flexible mechanism for describing entities in way that many different data publishers (located across the Internet) can add arbitrary information about the same entity, or create links between disparate entities.

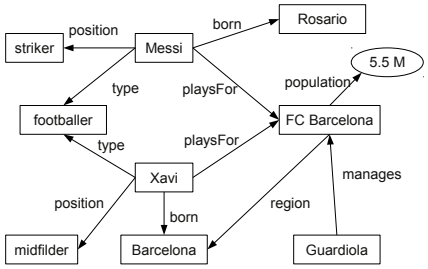


Figure 1: Example RDF Graph Data from DBpedia.

RDF uses a graph data model, where different entities are vertexes in the graph and relationships between them are represented as edges. Information about an entity is represented by directed edges emanating from the vertex for that entity (labeled by the name of the attribute), where the edge connects the vertex to other entities, or to special "literal" vertexes that contain the value of a particular attribute for that entity.

Figure 1 shows a sample RDF graph, taken from the famous DBpedia RDF dataset[1]. For example, edges in the graph indicate that the entity ("Messi") is of type "footballer", was born in Rosario, and plays striker for FC Barcelona. Each of the entities that "Messi" is connected to in this graph can have their own set of connections; for example, FC Barcelona is shown to be connected to the Barcelona entity through the region relation.

Most RDF-stores (systems that are built to store and query RDF data) represent RDF graphs as a table of triples, where there is one triple for each edge in the RDF graph. The triple takes the form $< subject, predicate, object >$, where the subject is the entity from which the (directed) edge emanated, the predicate is the label of the edge, and the object is the name of the entity or literal on the other side of the edge. Many RDF-stores store the triples in a relational database, using a single relational table containing three columns. The triple table for the example RDF graph in Figure 1 can be found in the appendix (Section A).

2.2 SPARQL

As explained in the official W3C Recommendation documentation for SPARQL [3], "Most forms of SPARQL queries contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples except that each of the subject, predicate and object may be a variable. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables." Hence, executing SPARQL queries generally involves graph pattern matching. For example, the query in Figure 2 returns the managers of all football (soccer) clubs in Barcelona. This example tries to find entities in the data set that have at least two edges emanating from them: one that is labeled "type" and connects to the "footballClub" entity, and one that is labeled "region" and connects to the "Barcelona" entity. Entities that match this pattern are referred to using the variable name ?club (the ? character is used to indicate pattern variables). For these entities, we look for edges that connect to them via the "manages" label and return both sides of that edge if such an edge exists.

[1]We made some simplifications to the actual DBpedia data in order to make this example easier to read. Most notably, we replaced the Universal Resource Identifiers (URIs) with more readable names.

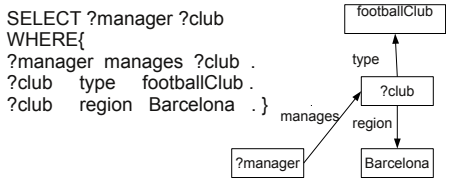


Figure 2: Example SPARQL Query.

If the RDF data is stored using a relational "triples" table, SPARQL queries can be converted to SQL in a fairly straightforward way, where the triples table is self-joined for each SPARQL condition, using shared variable names as the join equality predicate. For example, the SPARQL query above can be converted to the following SQL:

```
SELECT A.subject, A.object
FROM   triples AS A, triples AS B, triples AS C
WHERE B.predicate = "type" AND B.object = "footballClub"
  AND B.subject = C.subject  AND  C.predicate = "region"
  AND C.object = "Barcelona"  AND  C.subject = A.object
  AND A.predicate = "manages"
```

A more complicated SPARQL query is presented as Example 1 in the appendix.

In general, SPARQL graph patterns that involve paths through the RDF graph convert to subject-object joins in the SQL, and patterns that involve multiple attributes about the same entity involve subject-subject joins in the SQL (the above example has both types of joins). Although other types of joins are possible, subject-subject and subject-object joins are by far the most common.

3. SYSTEM ARCHITECTURE

As described in Section 2.2, SPARQL queries tend to take the form of subgraph matching. This type of data access is therefore the motivating use-case for which we are architecting our system. RDF data is partitioned across a cluster of machines, and SPARQL graph patterns are searched for in parallel by each machine. Data may need to be exchanged between machines in order to deal with the fact that some patterns in the data set may span partition boundaries. We use Hadoop to manage all parts of query processing that require multiple machines to work together.

The high level architecture diagram is presented in Figure 3. RDF triples that are to be loaded into the system get fed into the data partitioner module which performs a disjoint partitioning of the RDF graph by vertex. As will be described in the next section, we default to using a popular open source graph partitioner that runs on a single machine (our master node); for RDF datasets that are too large to be partitioned by a single machine, we can plug in a distributed graph partitioning solution instead. The output of the partitioning algorithm is then used to assign triples to worker machines according to the triple placement algorithm we will describe in the next section. Each worker machine contains an installation of a pluggable state-of-the-art RDF-store, and loads all triples it receives from the triple placer.

In order to increase the number of queries that are possible to be run completely in parallel, it is beneficial to allow some triples to be replicated on multiple machines. The data replicator on each worker node determines which triples are on the boundary of its partition, and replicates these triples

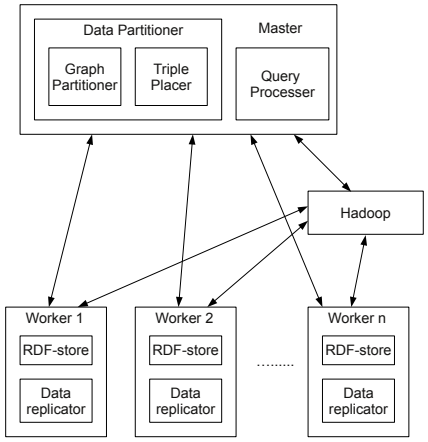


Figure 3: System Architecture.

(according to the n-hop guarantee we present later) using a Hadoop job after each new batch of triples are loaded.

The master node also serves as the interface for SPARQL queries. It accepts queries and analyzes them closely. If it determines that the SPARQL pattern can be searched for completely in parallel by each worker in the cluster, then it sends the pattern to each node in the cluster for processing. If, however, it determines that the pattern requires some coordination across workers during the search, it decomposes the query into subgraphs that can be searched for in isolation, and ships these subgraphs to the worker nodes. It then hands off the rest of the query, including the query processing needed to reconstruct the original query graph, to Hadoop to complete.

The following sections describe the load and query steps in detail. We assume that data is read-only after being loaded (updates are left for future work).

4. DATA PARTITIONING

When data is partitioned across multiple machines, the particular partitioning algorithm can make a large difference in the amount of data that needs to be shipped over the network at query time. Take the following example that returns the names of the strikers that have played for FC Barcelona:

```
SELECT ?name
WHERE  {  ?player type      footballPlayer .
          ?player name      ?name          .
          ?player position  striker        .
          ?player playsFor  FC_Barcelona   . }
```

If data was partitioned using a simple hash partitioning algorithm, it is trivial to perform it in parallel without any network communication (except at the very end of the query to collect results). In particular, if data was hash partitioned by RDF subject (which, as described above, is how clustered RDF-store implementations are generally designed), then it is guaranteed that all triples related to a particular player are stored on the same machine. Therefore, every machine in the cluster can process the query (completely in parallel) for the subset of players that had been hash partitioned to that machine, and the results can be aggregated together across machines at the end of the query.

The type of query that hash partitioning by subject excels at are SPARQL queries that contain a graph pattern forming a "star", with a single center vertex (that is usually a

variable), and one or more edges emanating from this center vertex to variable or constant vertexes. The above example is exactly this type of query.

While star queries are fairly common in SPARQL workloads, most benchmarks and realistic workloads contain additional queries that attempt to match more complicated SPARQL patterns. Consider Figure 2 from Section 2.2. The query forms a short path (of length two) through the RDF data. No matter how the data is hash partitioned across machines, the query cannot be performed completely in parallel without network communication because the three patterns do not share a common subject, predicate nor object. Consequently, data shuffling will be necessary at query time, and if any pattern of the query happens to be unselective, the network may be overwhelmed by intermediate results.

We therefore choose to graph partition RDF data across machines instead of simple hash partitioning by subject, predicate, or object. This allows vertexes that are close together in the RDF graph to be stored on the same machine (unless they are near a partition boundary) and graph patterns that contain paths through the graph of length two or above can mostly be performed in parallel (as long as some correction is taken for RDF graph subsets on the partition boundaries that could span multiple machines).

Since every RDF triple describes a particular edge in the RDF graph (i.e., the number of RDF triples in a data set is equal to the number of edges in a RDF graph), the most obvious way to perform graph partitioning is via "edge-partitioning" a RDF graph into disjoint subsets. Unfortunately, such an approach complicates the execution of star queries since vertexes on the boundary of a partition might have some edges on one machine and some edges on another machine, thereby causing additional network communication at query time relative to simple hash partitioning.

Therefore, since star queries are quite common, we partition a RDF graph by vertex, such that each machine in a cluster receives a disjoint subset of RDF vertexes that are close to each other in the graph. Once the RDF graph has been partitioned, we assign triples to machines. The most obvious way to do this is to place each triple on the machine that owns the subject vertex for that triple; however, we will discuss some alternatives to this approach below.

Our graph partitioning scheme consists of two major steps. We first divide vertexes of an RDF graph into disjoint partitions. Then we assign triples to partitions based on the original RDF graph, the output of vertex partitioning, and other parameters. We discuss in depth these two steps in the next two sections.

4.1 Vertex Partitioning

To facilitate partitioning a RDF graph by vertex, we remove triples whose predicate is rdf:type (and other similar predicates with meaning "type"). These triples may generate undesirable connections, because if we included these "type" triples, every entity of the same type would be within two hops of each other in the RDF graph (connected to each other through the shared type object). These connections make the graph more complex and reduce the quality of graph partitioning significantly, since the more connected the graph is, the harder it is to partition it. In fact, we find that removing the bias to place objects of the same type on the same machine can sometimes improve query performance, since this causes entities of popular types to be spread out across the cluster, which allows queries involving these types to be executed in parallel in an equitable fashion[2].

Vertex partitioning of graphs is a well-studied problem in computer science, and we therefore can leverage previously existing code to do the partitioning for us. In our current prototype, we use the METIS partitioner [1] for this purpose; however, we expect to switch to a more scalable partitioner in the future. We input the RDF graph as an undirected graph to METIS, specify the desired number of partitions (which is usually the number of machines in the cluster), and METIS outputs partitions of vertexes that are pairwise disjoint.

4.2 Triple Placement

After partitioning the vertexes of the RDF graph into disjoint partitions, the next step is to assign triples from the RDF data set into these partitions. As mentioned above, the most straightforward way to implement this is to simply check the subject for each triple and place the triple in the partition that owns the vertex corresponding to that subject.

It turns out, however, that allowing some triples (particularly those triples whose subject is on the partition boundary) to be replicated across partitions can significantly reduce the amount of network communication needed at query time to deal with the fact that SPARQL query patterns can potentially span multiple partitions. Of course, this leads to a fundamental trade-off between storage overhead and communication overhead. Less triple replication reduces storage overhead (and also I/O time if a query requires that all data per partition needs to be scanned) but incurs communication overhead. More replication results in higher storage overhead, but it cuts down the communication cost, and it can, in some cases, make the query executable completely in parallel across partitions without any additional communication.

Since we are using Hadoop as the query execution layer in our system to do the coordination of query processing across machines, there can be a dramatic performance difference between queries that can be processed completely in parallel without any coordination (where the use of Hadoop can be circumvented) and queries that require even a small amount of coordination (where Hadoop needs to be fired up). This is because Hadoop usually has at least 20 seconds of start-up overhead, so query time can increase from a small number of seconds (or even less) for completely parallelizable queries, to at least 20 seconds for queries that require data exchange.

To specify the amount of overlap, we define a *directed n-hop guarantee* as follows: For a set of vertexes assigned to a partition, a directed 1-hop guarantee calculates all vertexes in the complete RDF graph that are connected via a single (directed) edge from any vertex already located within the partition, and adds this set of vertexes along with the edges that connect them to the original set of vertexes in the partition. A directed 2-hop guarantee starts with the subgraph that was created by the directed 1-hop guarantee and adds another layer of vertexes and edges connected to this new subgraph. This works recursively for any number of hops. (A formal definition of the directed n-hop guarantee is given

[2]Similar logic applies for why parallel databases tend to partition large tables equitably across a cluster instead of trying to fit each table onto a single machine.

in Section B in the appendix). The triples corresponding to the edges added by the directed n-hop guarantee are placed at that partition (even if they already exist elsewhere). Note that the directed 1-hop guarantee yields no triple replication (it is the same as the "straightforward" triple placement algorithm mentioned above).

A potential problem with using a *directed* n-hop guarantee is that vertexes that correspond to an *object* in one RDF triple and as a *subject* or *object* in another triple (and that exist on the edge of a partition) can potentially result in their associated triples being allocated to different partitions (without replication). For example, triples (s, p, o) and (o, p', o') where the subject of one triple is equal to the object of another triple are not guaranteed to end up in same the partition if only one hop is guaranteed (however they will end up in the partition that "owns" vertex s under a 2-hop guarantee). Furthermore, triples (s, p, o) and (s', p', o) are not guaranteed to end up in the same partition (even though they are connected via the same object) no matter how large the n-hop guarantee is.

Since it is not unusual for a SPARQL query to include such an "object-connected" graph pattern, in many cases it is preferable to use an *undirected* n-hop guarantee, so that triples connected through an object will end up in the same partition. (A formal definition of the undirected n-hop guarantee is given in Section C in the appendix). For instance, the SPARQL query in Figure 2 can be performed completely in parallel if data is partitioned using an undirected 1-hop guarantee (the whole query pattern is within one undirected hop from the ?club vertex). Each partition performs the subgraph match for the clubs that are owned by that node, knowing that all information about the manager and city of each club can be found within one (undirected) hop of that club and are therefore stored in the same partition. An undirected hop guarantee results in additional triple replication (relative to a directed hop guarantee), but it is able to yield improved performance on a wider range of queries[3].

We implement the triple placement algorithm (whether directed or undirected) in a scalable fashion using Hadoop jobs. The original RDF graph, along with the initial assignment of vertexes to partitions (from the graph partitioning algorithm) are partitioned across the Hadoop cluster. For each hop in the n-hop guarantee, a Map phase iterates through all triples in order to generate a list of vertexes that are directly connected with each other, and sends this list along with the current vertex-to-partition mapping to a Reduce phase which receives, for each vertex, both its direct connections and also the set of nodes it is currently mapped to, and adds these direct connections to these set of nodes (thereby supplementing the current vertex-to-partition mapping). MapReduce pseudocode for triple placement is given in Figure 13 in the appendix.

For both the directed and undirected n-hop guarantees, in order to avoid returning duplicate results due to the replication of triples across partitions, we distinguish between triples that are owned by a partition (i.e., the subject of these triples is one of the base vertexes of that partition) and triples that were replicated to that partition to satisfy the n-hop guarantee. We do this by explicitly storing the list of base vertexes on each machine. To unify the format, we store this list of base vertexes in triple format. For each base vertex v in the vertex partition, we add a triple (v, '<isOwned>', 'Yes'). We will show in Section 5 that representing this list as triples will simplify query execution.

Triple patterns involving rdf:type appear very frequently in SPARQL queries; therefore, for each vertex that is added in the last round of the n-hop guarantee generation, we also add the triple emanating from this vertex containing its rdf:type if such a triple exists (even though this is technically an additional hop). The overhead for this is typically low, and this further reduces the need for cross-partition communication.

High degree vertexes (i.e., vertexes that are connected to many other vertexes) cause problems for both the regular graph partitioning scheme (since well connected graphs are harder to partition) and for the n-hop guarantee (since adding these vertexes causes many edges to be dragged along with them). One potential optimization to avoid this problem is to ignore these high degree vertexes during the partitioning and n-hop guarantee triple placement steps. Before the graph partitioning, the data would be scanned (or sampled) in order to find the degree of each vertex, and then the average degree for each type (rdf:type) is calculated. If the average exceeds a threshold (the default is three standard deviations away from the average degree across all types), vertexes from that type are excluded from graph partitioning (and therefore all triples whose subject is equal to a vertex from that type are removed). After partitioning, the dropped vertexes are added to the partitions based on which partition contains the most edges to that vertex.

During triple placement, high-degree vertexes bring far more triples into the hop guarantee than other vertexes do. The overlap between partitions grows exponentially when high-degree vertexes are connected to each other but assigned to different partitions. A similar optimization can help with this problem as well: the hop guarantee can be weakened so that it does not include triples of high-degree types (which is defined identically to the previous paragraph). The query processing algorithm must be made aware that triples of high-degree types are not included in the hop guarantee, and graph patterns involving these types require additional communication across partitions.

5. QUERY PROCESSING

In our system, queries are executed in RDF-stores and/or Hadoop. Since query processing is far more efficient in RDF-stores than in Hadoop, we push as much of processing as possible into RDF-stores and leave the rest for Hadoop. Data partitioning has a big impact on query processing. The larger the hop guarantee in the data partitioning algorithm, the less work there is for Hadoop to do (in some cases, Hadoop can be avoided entirely).

If a query can be answered entirely in RDF-stores without data shuffling in Hadoop, we call it a "PWOC query" (parallelizable without communication). Figure 4 sketches the algorithm to determine whether a query is PWOC given an n-hop guarantee. The concept of "distance of farthest edge" (DoFE) in the algorithm is a measure of centrality in a graph. Intuitively, the vertex in a graph with the smallest DoFE will be the most central in a graph. We will call this

[3]Of course, there are plenty of queries for which an undirected hop guarantee provides no additional benefit over a directed hop guarantee. For example, Figure 1 from the appendix can be performed completely in parallel if data is partitioned using any kind (directed or undirected) of two-hop guarantee.

This algorithm assumes an undirected n-hop guarantee and the input query $G = \{V, E\}$ is an undirected graph. Algorithms for directed hop guarantees and incomplete hop guarantees (high degree vertex removal) are given in Figures 14 and 15 in the appendix.

function **IsPWOC**(Input: G, n, Output: Boolean)
core=v, *s.t.* DoFE$(v, G) \leq$DoFE(v', G), $\forall v' \in V$
return (DoFE(core, G) $\leq n$)

function **DoFE**(Input: G, Vertex v, Output: Int)
$\forall e = (v_1, v_2) \in E$, compute
min(distance(v, v_1), distance(v, v_2)) + 1,
denoted by dist(v, e)
return dist(v, e) *s.t.* dist$(v, e) \geq$ dist(v, e') $\forall e' \in E$

Figure 4: Determining whether a Query is PWOC (Parallelizable Without Communication.)

vertex the "core" in function **IsPWOC**.

Since partitions overlap in the triples they own, the issue of duplicate results in query processing needs to be addressed. One naive approach to deal with this problem would be to use a Hadoop job to remove duplicates after the query has completed. Instead, our system adopts an owner-computes model [21] popular in parallel graph computation. Recall from Section 4.2 that we add triples (v, '<isOwned>', 'Yes') to a partition, if v was one of the "base" vertexes assigned to that partition by the disjoint partitioning algorithm. For each query issued to the RDF-stores, we add an additional pattern (*core*, '<isOwned>', 'Yes'), where *core* is found in the function **IsPWOC**. In this way, each partition only outputs the subgraph matches whose binding of the core is owned by the partition.

If a query is not PWOC, we decompose the query into PWOC subqueries. We then use Hadoop jobs to join the results of the PWOC subqueries. The number of Hadoop jobs required to complete the query increases as the number of subqueries increases. Therefore, our system adopts the query decomposition with the minimal number of subqueries as a heuristic to optimize performance. This reduces to the problem of finding minimal edge partitioning of a graph into subgraphs of bounded diameter, a problem that has been studied well in the theory community [8, 6, 20, 14]. Although more optimal algorithms are known, we use a brute-force implementation in our prototype, since the SPARQL graphs we have worked with contain 20 or fewer triple patterns.

Take, for example, the example SPARQL query presented in Section 2.2 (Figure 2). For this example, the DoFEs for manager, footballClub, Barcelona and club are 2, 2, 2 and 1, respectively (if we are using an undirected hop guarantee). club will be chosen as the core because it has the smallest DoFE. Another SPARQL query is presented in Example 1 in the appendix. For that query, the DoFEs for footballer, pop, region, player and club are 3, 3, 2, 2 and 2, respectively if we are using an undirected hop guarantee. Any of region, player or club may be chosen as the core. If we are using a directed hop guarantee, only player has a DoFE of 2. Suppose our data partitioning uses an undirected 1-hop guarantee. Figure 2 will be considered a PWOC query because the DoFE for club is the same as the hop guarantee. A triple patten (?*club*, '<isOwned>', 'Yes') will be added to the query which is then issued to the RDF-stores. However, Example 1 is not a PWOC query and therefore decomposed into PWOC subqueries (see Section D in the appendix for the resulting subqueries). Player and region are chosen as cores in the subqueries because they both have DoFE 1. After the two subqueries are executed in the RDF-stores, the intermediate results are joined on club and region by a Hadoop job. For a directed or undirected 2-hop guarantee, both examples are PWOC queries.

In general, the above described query decomposition only needs to be done to check for graph patterns that could potentially span multiple partitions. However, even when the hop guarantee is not large enough to make the entire query PWOC, the internal vertexes for a partition (i.e. nodes that are not along the edge of a partition, which have more hops within local reach than the hop guarantee) can be checked directly without decomposition (and extra network communication) as if the query were PWOC. Section F.3 in the appendix explains this optimization further and experimentally studies its effects.

6. EXPERIMENTS

In this section, we measure the performance of our system on the Lehigh University Benchmark (LUBM) (the most widely used benchmark of the Semantic Web community). For comparison, we also measure the performance of a simple hash partitioning version of our system, another horizontally scalable RDF-store (SHARD[27], which also uses hash partitioning), and RDF-3X[24] running by itself on a single node. The specifications of our 20-node cluster, along with some additional technical details and background on RDF-3X and SHARD, are given in Appendix E.

We will refer to the prototype of the system described in this paper as "graph partitioning". As mentioned in Section 3, our system involves installing a fast RDF store on each machine, attempting to parallelize SPARQL queries over RDF data partitioned across these machines as much as possible, and using Hadoop to manage any redistribution of data that needs to occur at query time. For Hadoop, we used version 0.19.1 running on Java 1.6.0. We kept the default configuration settings. For the state-of-the-art RDF store, we used RDF-3X 0.3.5. This allows for a direct comparison of RDF-3X containing all data running on a single node, and a parallelized version of RDF-3X that we introduced in this paper. In this section, we experiment with both undirected 1-hop and 2-hop guarantees, with the high-degree vertex optimization described at the end of in Section 4.2 turned on. In Appendix F, we also experiment with directed hop guarantees, how performance scales with cluster size, and measure the contribution to performance of various optimizations.

The setup for the "hash partitioning" version of our system that we experiment with is identical to "graph partitioning" above (many RDF-3X nodes sitting underneath Hadoop), except that the data is hash partitioned by RDF subject instead of using the graph partitioning algorithm described in this paper. These results are included in the experiments to approximate the "best case scenario" for modern hash-partitioned scalable RDF-stores if they had used storage optimized for RDF (which is not the case today).

6.1 Benchmark

The Lehigh University Benchmark (LUBM)[17] features an ontology for the university domain, synthetic OWL and RDF data scalable to an arbitrary size, and fourteen exten-

sional queries representing a variety of properties. In these experiments, we generate a dataset with 2000 universities. The data size is around 50 GB in N-Triples format and contains around 270 million triples. As a measurement of query complexity, Figure 8 in the appendix shows the number of (self-)joins in each query of the benchmark. In a nutshell, out of the 14 queries, 12 have joins, of which all 12 have at least one subject-subject join, and 6 of them also have at least one subject-object join.

6.2 Data Load Time

Figure 5 shows the load time for the four systems. Hash partitioning takes the least time because it requires minimal pre-processing and loads in parallel. Graph partitioning takes 4 hours and 10 minutes in total, most of which is spent on triple placement. Loading into RDF-3X is much faster for graph partitioning than loading it all on a single node since each node only needs to load a little more than 1/20th as much data, and RDF-3X's load time does not scale linearly. SHARD's high load time is due to format conversion, and could probably be optimized with additional effort.

System	Load Time
RDF-3X	2.5h
SHARD	6.5h
Hash Partitioning	0.5h
Graph Partitioning	Vertex Partitioning: 1h Triple Placement: 3h Loading into RDF-3Xs: 10min

Figure 5: Load Time.

6.3 Performance Comparison

Figure 6 shows the execution time for LUBM in the four benchmarked systems. Except for query 6, all queries take more time on SHARD than on the single-machine deployment of RDF-3X. This is because SHARD's use of hash partitioning only allows it optimize subject-subject joins. Every other type of join requires a complete redistribution of data over the network within a Hadoop job, which is extremely expensive. Furthermore, its storage layer is not at all optimized for RDF data (it stores data in flat files).

To analyze the performance of our system, we first divide the queries into two groups. Queries 1, 3, 4, 5, 7, 8, 10, 11 and 12 run for less than 1 second on single-machine RDF-3X, so we call them "fast queries"; queries 2, 6, 9, 13 and 14 run for more than 10 seconds on single-machine RDF-3X, so we call them "slow queries".

For fast queries (queries that require little more than a few index lookups), the large size of the data set is not problematic since the data size is significantly reduced before a scan is required, so adding more machines and partitioning data across them does not improve the performance. Therefore, it is slightly better to run these queries on a single machine than in parallel on multiple machines, due to the network communication delay needed to get these parallel queries started and aggregate results. These "fast" queries take a minimum 0.4 to 0.5 seconds on the multi-machine implementations just for this communication.

For slow queries (queries that require scans of large amounts of intermediate data), the single machine implementation sees significant scalability limitations, and partitioning the data across multiple machines dramatically improves performance. The improvement ranges from 5 times (query 9) to 500 times (query 2) faster. The reason for the super-linear

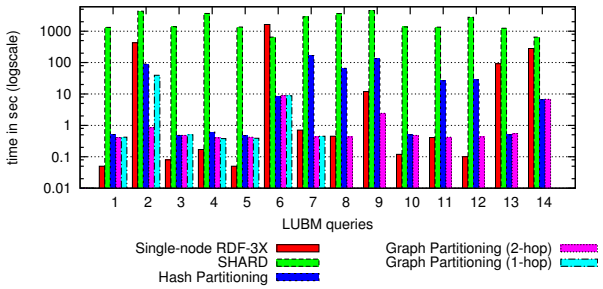


Figure 6: LUBM Execution Time.

speedup is explained further in Section F.1 (in short, partitioning allows the working set to be held in memory rather than on disk).

Changing the hop guarantee for the graph partitioning technique from 1-hop to 2-hop only has a significant effect for three queries (2, 8 and 9) since the 1-hop guarantee is sufficient to make most of the queries PWOC on this benchmark. This is explored further (along with reports on data sizes after triple replication) in Section F.2 in the appendix. The 2-hop guarantee is sufficient to turn all joins into PWOC joins, since no SPARQL query in the benchmark has a graph pattern diameter larger than 4. This allows all queries to be performed completely in parallel for the 2-hop graph partitioning implementation.

Comparing graph and hash partitioning allows one to directly see the benefit of our proposed graph partitioning technique. Hash partitioning is only able to avoid network communication for queries containing only subject-subject joins (or no joins at all). However, for queries with subject-object joins, the hash partitioning approach has to ship data across the network to perform the join, which adds significant latency for data redistribution and Hadoop processing. Additional subject-object joins result in additional joins, which result in further slowdown. For instance, queries 8, 11 and 12 have only one subject-object join and take up to 90 seconds; while queries 2, 7 and 9 have two or three subject-object joins and take up to 170 seconds.

For queries with no subject-object joins (1, 3, 4, 5, 6, 10, 13 and 14), where all joins are PWOC joins for both hash partitioning and graph partitioning, hash partitioning sometimes performs slightly better than graph partitioning, since the graph partitioning algorithm requires an additional join per query to filter out subgraph matches that are centered on a vertex that is not a base vertex of a partition (the technique that removes duplicate results).

Note the log-scale of Figure 6. Running the benchmark takes SHARD 1340 times longer than our graph partitioning implementation.

7. RELATED WORK

Pregel[22] is a parallel graph computation framework. A typical Pregel computation consists of a sequence of supersteps separated by global synchronization points when messages are passed between machines (similar to how we divide work between individual databases and cross-node communication inside Hadoop). Pregel does not use the multi-hop guarantees we use in this paper to improve parallelism.

Parallel Boost Graph Library(PBGL)[16] and CGMgraph [10] optionally implement something similar to a 1-hop guarantee. For example, PBGL is a C++ library for distributed graph computation. It includes the concept of bidirectional

graphs that require that the set of edges incoming to a given vertex are accessible in constant time, in addition to the outgoing edges (which is essentially the same as the undirected 1-hop guarantee presented in this paper). However, this previous work does not generalize the hop-guarantee, implement any of the optimizations that we present in this paper, nor present an algorithm for maximizing the parallelization of queries given a general hop-guarantee.

SUBDUE[13] is a knowledge discovery system that also uses something akin to a one-hop guarantee. Since it is a KDD system, it does not guarantee accurate output.

Our work on searching for patterns in graphs leverages previous work on pushing this effort into database systems [28]. Our join-based approach for pattern matching is related to work by Cheng et. al. [11] and our distance calculations are related to recent work by Zou et. al. [34].

Weaver et. al. [31] present an approach for calculating graph closure under RDFS semantics using parallel computation. The work by Urbani et. al. [30] works similarly, but uses MapReduce to perform this calculation. Neither work focuses on processing general SPARQL queries.

Several attempts have been made to use MapReduce for SPARQL querying [27, 23]. As we showed in the experiments, these techniques are unable to leverage graph partitioning and have storage layers that are not optimized for RDF data. Therefore, these systems, while horizontally scalable, are quite inefficient.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a scale-out architecture for scalable RDF data management that leverages state-of-the-art RDF-stores and the popular MapReduce framework. Based on the graph nature of RDF data, we proposed a graph-oriented data partitioning scheme to exploit the spacial locality inherent in graph pattern matching. By pushing most or even all of query processing into fast single-node RDF stores which operate in parallel, our system is able to perform up to three orders of magnitude faster than other attempts at horizontally scalable RDF data management. For future work, we plan to add a workload-aware component to our graph partitioning technique, and explore how updates can be handled efficiently by our architecture.

9. ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under Grant Number IIS-0845643. Kun Ren is supported by National Natural Science Foundation of China under Grant 61033007.

10. REFERENCES

[1] METIS. http://glaros.dtc.umn.edu/gkhome/views/metis.

[2] RDF Primer. W3C Recommendation. `http://www.w3.org/TR/rdf-primer`, 2004.

[3] SPARQL Query Language for RDF. W3C Working Draft 4 October 2006. `http://www.w3.org/TR/rdf-sparql-query/`, 2006.

[4] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, Vienna, Austria, 2007.

[5] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[6] I. Abraham, C. Gavoille, D. Malkhi, and U. Wieder. Strong-diameter decompositions of minor free graphs. SPAA '07, pages 16–24, 2007.

[7] S. Alexaki, V. Christophides, G. Karvounarakis, K. Tolle, and D. Plexousakis. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *SemWeb*, 2001.

[8] B. Awerbuch and D. Peleg. Sparse partitions. In *In IEEE Symposium on Foundations of Computer Science*, pages 503–513, 1990.

[9] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC*, pages 54–68, 2002.

[10] A. Chan and F. Dehne. Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters. In *International Journal of High Performance Computing Applications*, pages 81–97, 2003.

[11] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922. IEEE, 2008.

[12] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *VLDB*, pages 1216–1227, 2005.

[13] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin. Approaches to parallel graph-based knowledge discovery. *Journal of Parallel and Distributed Computing*, 61, 2001.

[14] J. Fox and B. Sudakov. Decompositions into subgraphs of small diameter. *Comb. Probab. Comput.*, 19:753–774.

[15] Franz Inc. AllegroGraph 4.2 Introduction. `http://www.franz.com/agraph/support/documentation/v4/agraph-introduction.html`.

[16] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *POOSC*, 2005.

[17] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[18] S. Harris and N. Gibbins. 3store: Efficient bulk RDF storage. In *In Proc. of PSSS'03*, pages 1–15, 2003.

[19] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: a federated repository for querying graph structured data from the web. ISWC'07/ASWC'07, pages 211–224, 2007.

[20] N. Linial and M. Saks. Low diameter graph decompositions. *Combinatorica*, 13:441–454, 1993.

[21] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[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] J. Myung, J. Yeon, and S.-g. Lee. Sparql basic graph pattern processing with iterative mapreduce. MDAC '10, pages 6:1–6:6.

[24] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19:91–113, 2010.

[25] Ontotext. OWLIM Benchmarking: LUBM. `http://www.ontotext.com/owlim/lubm.html`.

[26] OpenLink Software. Towards Web-Scale RDF. `http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticleWebScaleRDF`.

[27] K. Rohloff and R. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. *International Workshop on Programming Support Innovations for Emerging Distributed Applications*, 2010.

[28] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[29] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.

[30] J. Urbani, S. Kotoulas, E. Oren, and F. Harmelen. Scalable distributed reasoning using mapreduce. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 634–649, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] J. Weaver and J. A. Hendler. Parallel materialization of the finite rdfs closure for hundreds of millions of triples. ISWC '09, pages 682–697, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1:1008–1019, August 2008.

[33] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[34] L. Zou, L. C. 0002, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, pages 886–897, 2009.

APPENDIX

A. EXAMPLE DATA AND SPARQL QUERY

Figure 1 in Section 2 showed an example RDF graph. This graph can be converted to triple format (for storage in a typical RDF-store) in the following way:

subject	predicate	object
Lionel Messi	type	footballer
Lionel Messi	playsFor	FC Barcelona
Lionel Messi	born	Rosario
Lionel Messi	position	striker
Xavi	type	footballer
Xavi	playsFor	FC Barcelona
Xavi	born	Barcelona
Xavi	position	midfilder
FC Barcelona	region	Barcelona
Barcelona, Spain	population	5,500,000
Josep Guardiola	manages	FC Barcelona

EXAMPLE 1. *The following SPARQL query (that can be run over the above dataset) attempts to find football players playing for clubs in a populous region (population of at least 2 million) that also happens to be the region where he was born. (Such a player might have significant advertising value as a product spokesman in that region.) The graph version of the query is shown in Figure 7.*

```
SELECT ?player ?club ?region
WHERE  {  ?player type       footballer .
          ?player playsFor   ?club      .
          ?player born       ?region     .
          ?club   region     ?region     .
          ?region population ?pop      .
          FILTER (?pop > 2,000,000)  }
```

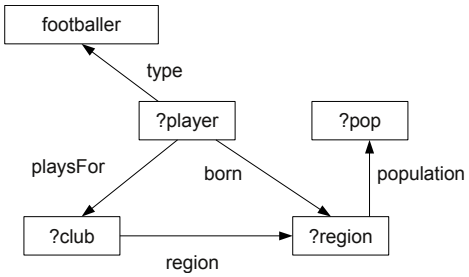


Figure 7: Example 1 in a Graph.

B. FORMAL DEFINITION OF THE DIRECTED N-HOP GUARANTEE

DEFINITION 1. *Let $G = \{V, E\}$ be a graph; and $W \subset V$. P_n, the directed n-hop guarantee for W, $P_n \subset G$ is defined recursively as follows:*
(1) $P_0 = \{V_0, E_0\}$ where $V_0 = W$ and $E_0 = \emptyset$ is the directed 0-hop guarantee for W.
(2)If $P_n = \{V_n, E_n\}$ is the directed n-hop guarantee for W, then $P_{n+1} = \{V_{n+1}, E_{n+1}\}$ is the directed (n+1)-hop guarantee for W where
V_{n+1}=$\{v|(v', v) \in E,\ v' \in V_n\} \cup V_n$,
E_{n+1}=$\{(v, v')|(v, v') \in E,\ v \in V_n, v' \in V_{n+1}\} \cup E_n$.

C. FORMAL DEFINITION OF THE UNDIRECTED N-HOP GUARANTEE

DEFINITION 2. *Let $G = \{V, E\}$ be a graph; and $W \subset V$. P_n, the undirected n-hop guarantee for W, $P_n \subset G$ is defined recursively as follows:*
(1) $P_0 = \{V_0, E_0\}$ where $V_0 = W$ and $E_0 = \emptyset$ is the undirected 0-hop guarantee for W [4].
(2)If $P_n = \{V_n, E_n\}$ is the undirected n-hop guarantee for W, then $P_{n+1} = \{V_{n+1}, E_{n+1}\}$ is the undirected (n+1)-hop guarantee for W where
V_{n+1}=$\{v|(v, v') \in E$ *or* $(v', v) \in E,\ v' \in V_n\} \cup V_n$,
E_{n+1}=$\{(v, v')|(v, v') \in E,\ v \in V_n, v' \in V_{n+1}\} \cup$
$\{(v', v)|(v', v) \in E,\ v \in V_n, v' \in V_{n+1}\} \cup E_n$.

D. QUERIES ISSUED TO RDF-STORES

Section 5 discussed how SPARQL queries are decomposed and sent in pieces to be performed in parallel by the RDF-store on each partition. The specific SPARQL that is sent to the RDF-stores for the two examples from that section are given here.

For the SPARQL query in Figure 2, the following query is issued to the RDF-stores:

```
SELECT ?manager ?club
WHERE  {  ?manager  manages  ?club        .
          ?club     type     footballClub .
          ?club     region   Barcelona    .
          ?club     isOwned  Yes          .}
```

For Example 1 from Appendix A, the query is decomposed into two subqueries:

```
Subquery 1:

SELECT ?player ?club ?region
WHERE  {  ?player type       footballer .
          ?player playsFor   ?club      .
          ?player born       ?region      .
          ?player isOwned    Yes        .}

Subquery 2:

SELECT ?club ?region
WHERE  {  ?club   region     ?region .
          ?region   population ?pop    .
          ?region   isOwned    Yes     .
          FILTER (?pop > 2,000,000)  }
```

E. MORE EXPERIMENTAL DETAILS

In this section we describe the configuration of our experimental cluster, and give some additional details about RDF-3X and SHARD which we use as comparison points in our experiments.

E.1 Experimental Setup

Except for the single-machine experiments, each of the systems we benchmark are deployed on a 20-machine cluster. Each machine has a single 2.40 GHz Intel Core 2 Duo

[4] Since the 0-hop guarantee contains no edges, it contains no triples, and is therefore never used in practice.

processor running 64-bit Red Hat Enterprise Linux 5 (kernel version 2.6.18) with 4GB RAM and two 250GB SATA-I hard disks. According to hdparm, the hard disks deliver 74MB/sec for buffered reads. All machines are on the same rack, connected via 1Gbps network to a Cisco Catalyst 3750E-48TD switch.

E.2 RDF-3X

RDF-3X[24] is a state-of-the-art single-node RDF-store. It builds indexes over all possible permutations of subject, predicate and object. These indexes are highly compressed and leveraged by the query processor to perform efficient merge joins (as described in Section 2.2, relational triple store implementations of RDF-stores require many self-joins). The query optimizer in RDF-3X employs a cost model based on RDF-specific statistics to make the optimal choice for join orders. Its recent release also supports online and batch updates. We use RDF-3X version 0.3.5 in our experiments.

E.3 SHARD

SHARD[27] is an open-source, horizontally scalable triple-store system. Its data processing and analytical frameworks are built using the Cloudera Distribution of the Hadoop implementation of the MapReduce formalism.

The system persists data in flat files in the HDFS file system such that each line of the triple-store text file represents all triples associated with a different subject. The query processing engine in SHARD iterates over the triple-store data for each clause in the input SPARQL query, and incrementally attempts to bind query variables to literals in the triple data, while satisfying all of the query constraints. Each clause of the SPARQL query is processed in a separate MapReduce operation.

For our experiments, we keep the default configurations of Hadoop and SHARD.

E.4 Number of Joins in LUBM Queries

As a measurement of query complexity, Figure 8 shows the numbers of joins, subject-subject joins (s-s joins) and subject-object joins (s-o joins) in each query of the benchmark if the query were to be executed using SQL over a relational DBMS using the standard triple-store representation of one table with three columns (one for each: subject, predicate, and object) discussed in Section 2. Each "join" is a self-join of the triples table.

Query	#joins	#s-s joins	#s-o joins
1	1	1	0
2	6	3	3
3	1	1	0
4	4	4	0
5	1	1	0
6	0	0	0
7	3	1	2
8	4	3	1
9	6	3	3
10	1	1	0
11	2	1	1
12	3	2	1
13	1	1	0
14	0	0	0

Figure 8: Number of Joins in the Queries.

F. ADDITIONAL EXPERIMENTS

In this section, we measure how performance scales as the number of machines in the cluster increases, experiment with directed hop guarantees, and measure the contribution to performance of various optimizations.

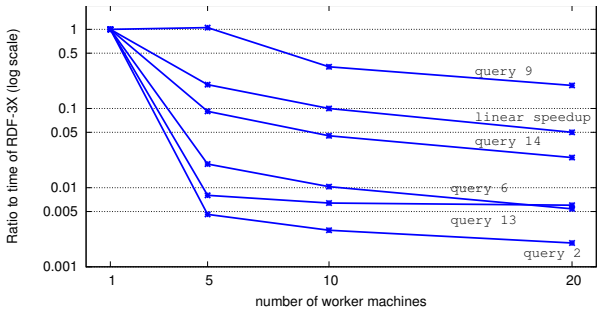
F.1 Varying Number of Machines

To explore the performance of our system as we vary the number of machines in the cluster, we run LUBM (2000) on clusters with 1, 5, 10, and 20 worker machines. We benchmark the optimized undirected 2-hop guarantee version of graph partitioning.

Figures 9 presents the results for the interesting queries[5]. We normalize execution time using the single-machine implementation as the baseline (all lines start at 1) and calculate the relative performance for the multi-machine cluster. We also include a line showing a theoretical system that achieves linear speedup (0.2, 0.1 and 0.05 for 5-, 10- and 20-machine clusters, respectively) for comparison. Except for query 9, all queries presented in this figure **exceed** linear speedup. This is because scaling out the cluster causes the working dataset on each machine to be small enough to fit into main memory, which dramatically reduces the needed disk I/O.

Query	RDF-3X	5 machines	10	20
2	428.77	1.98	1.25	0.84
6	1628.31	32.46	16.72	8.83
9	11.97	12.6	4.03	2.34
13	91.94	0.74	0.59	0.55
14	281.88	25.97	12.75	6.76

Query Performance in Raw Numbers.



After Normalization.

Figure 9: Speedup as More Machines are Added.

F.2 Parameter Tuning

To evaluate how different parameters influence our graph partitioning technique, we experiment in this section with the six parameter configurations listed in Table 1. "Un-two-hop-on" is the configuration we used in Sections 6 and F.1. For the sake of comparison, we also include the hash partitioning results in this section.

We first explore the impact of parameters on the data size. Figure 10 shows the triple counts and the normalized counts relative to hash partitioning (which has zero storage overhead since each triple exists in only one partition). The high-degree vertex optimization techniques do not make

[5]The response time for "fast queries" as defined in Section 6 is dominated by start-up costs and more machines do not improve the performance. We therefore call fast queries "uninteresting".

Name	Guarantee	High-degree Optimizations
un-two-hop-on	undirected-two-hop	on
un-two-hop-off	undirected-two-hop	off
un-one-hop-on	undirected-one-hop	on
un-one-hop-off	undirected-one-hop	off
dir-two-hop-off	directed-two-hop	off
dir-one-hop-off	directed-one-hop	off

Table 1: Parameter Configurations.

Configuration	Total Triples	Ratio to Hash
Hash Partitioning	276M	1
dir-one-hop-off	325M	1.18
un-1-hop-on	330M	1.20
un-1-hop-off	333M	1.21
dir-two-hop-off	329M	1.19
un-2-hop-on	338M	1.22
un-2-hop-off	1,254M	4.54

Figure 10: Total Triple Counts.

much of a difference when there is only a 1-hop guarantee, but make a dramatic contribution to reducing the exponential data explosion problem for the undirected 2-hop guarantee. The undirected 2-hop guarantee with the optimizations turned on only stores an extra 20% of triples per partition, but turning off the optimizations results in nearly a factor of 5 data explosion. Directedness also helps keeping the data size in check since fewer triples must be replicated through the n-hop guarantee.

We next explore how parameters influence query performance. Figure 11 presents the performance of each configuration on the LUBM benchmark. The first thing to note is that data explosion caused by unchecked hop guarantees is not only a storage size problem, but it also affects performance. This is because larger numbers of triples per partition causes the query processor to scan larger intermediate datasets which slows down performance (for the same reason why RDF-3X scales poorly if it is run on a single machine). This is best seen by comparing un-two-hop-off and un-two-hop-on, which use identical query plans, but un-two-hop-off needs to store many more triples per partition, which results in a significant slowdown in performance for the same "slow" queries that the single-node RDF-3X implementations struggled on in Section 6.3.

Reducing the undirected hop guarantee from two to one causes three queries (2, 8 and 9) to become non-PWOC. These queries are therefore slower than their counterparts in the two-hop guarantee, but they still outperform hash partitioning by up to 50% because one-hop still requires less data to be shipped over the network than hash partitioning. For the rest of the queries, un-one-hop-on/off and un-two-hop-on guarantees have similar performance.

Although a directed hop guarantee tends to store less data in each partition, it leads to more non-PWOC queries. Compared to the undirected one hop guarantee, the directed one hop guarantee has three more non-PWOC queries (7, 11 and 12) and hence much worse performance on these queries. For query 7, even the directed two hop guarantee fails to make it PWOC. Hence for this benchmark, undirected hop guarantees are clearly better. But for other benchmarks, especially where the high degree optimization is less effective (because high degree vertexes are accessed more frequently the query

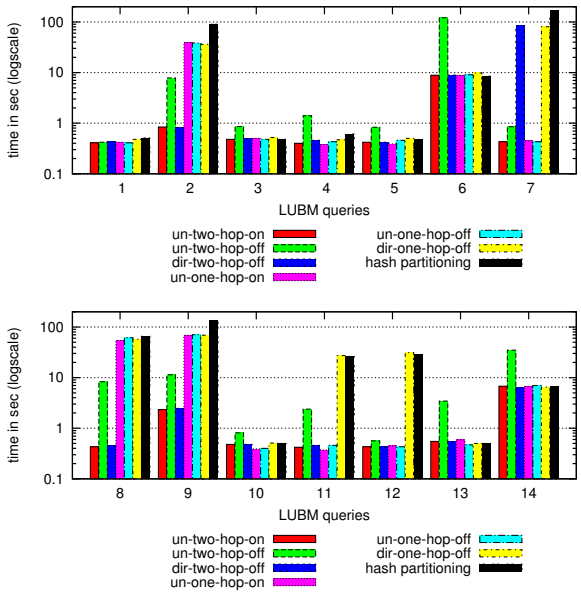


Figure 11: Performance of Different Configurations.

workload), a directed hop guarantee would likely perform better relative to a undirected guarantee.

F.3 Hop Radius Table

For queries where the hop guarantee is not large enough to make the query PWOC, it is still possible to reduce the network communication that occurs during query decomposition and execution in Hadoop. In general, each partition contains a subgraph of the original graph. Informally, a n-hop guarantee ensures that for each vertex in a partition, all vertexes that are reachable within n hops of that vertex in the original graph are stored locally on the same machine in which the partition resides (either because these vertexes were originally part of the partition, or because they were replicated over to that machine to satisfy the hop guarantee). Vertexes close to the boundary of the partition subgraph are likely not going to be able to rely on reaching vertexes farther away than the hop guarantee. However, the vertexes around the center of the subgraph may have access to complete sets of vertexes that are a more than n hops away. To distinguish these two different cases and quantify the position of a vertex in a subgraph, we define a hop radius.

DEFINITION 3. *Let $G = \{V, E\}$ be a graph, and P_n be the n-hop guarantee for W, P'_m be the m-hop guarantee for $\{v\}$, where $v \in W \subset V$ and $m \geq n$. If $P'_m \subseteq P_n$, then m is a hop radius for v in partition P_n.*

Informally, a hop radius is a hop guarantee for a *vertex* instead of a *partition*. We maintain a table on each partition that lists the hop radius for each vertex (which is always more than or equal to the hop guarantee). All vertexes in this table that has a hop radius larger than or equal to the DoFE of a query can be checked directly for a match between the SPARQL subgraph and a subgraph in the actual RDF data using that vertex as the "core". These vertexes can be checked completely in parallel, as if the query were PWOC. Only the remaining vertexes need to be checked via the query decomposition process described in Section 5.

We now explore the effects of this optimization the system. We only show results for non-PWOC queries here because

PWOC queries need not use a hop radius table. As can be seen in Figure 12, queries 8 and 9 perform roughly 20 seconds when a hop radius table is used to indicate which parts of the graph can be checked for subgraph matching completely locally on individual nodes, thereby reducing the intermediate results that need to be shipped over the network for the query decomposition algorithm. However, some queries are not bottlenecked by network communication, and therefore perform similarly with or without hop radius table (e.g. queries 11 and 12).

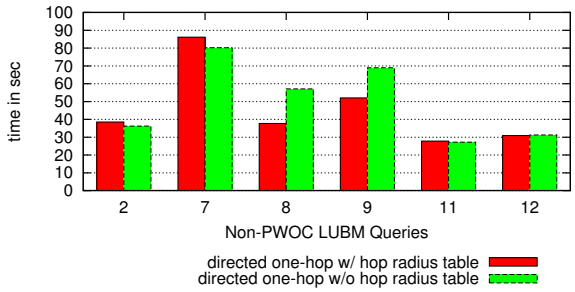


Figure 12: Improvement with Hop Radius Table.

G. TRIPLE PLACEMENT ALGORITHM

Let $G = \{V, E\}$ be input graph. Let R be the vertex to machine assignment, namely, a binary relation (V, N). n is the undirected hop guarantee.

function **TriplePlacement**(Input:edges E, binary relation R, Output: placement (E, N) with undirected n-hop guarantee)

```
B_0 = R
for i from 1 to n
    {A_i,B_i} = OneHopExpansion(E, B_{i-1})
return A_1 ∪ A_2 ∪ ... ∪ A_n
```

function **OneHopExpansion**(Input:edges E, binary relation (V, N), Output: binary relations A (E, N), B (V, N))

```
A = ∅, B = ∅
Map: Input: (v_1, v_2) OR (v, partition id)
    if input is (v_1, v_2)
        emit (v_1, (v_1, v_2)) and (v_2, (v_1, v_2))
    else
        emit (v, partition id)
Reduce: Input: (v, partition ids I ∪ edges E')
    for each (v_1, v_2) ∈ E' and each i ∈ I
        add ((v_1, v_2), i) to A
        if v_1 = v
            add (v_2, i) to B
        else
            add (v_1, i) to B

return {A, B}
```

Figure 13: Triple Placement Algorithm in MapReduce for Undirected Hop Guarantee.

H. PWOC ALGORITHMS

This algorithm assumes a directed n-hop guarantee and the input query $G = \{V, E\}$ is a directed graph.

function **IsPWOC**(Input: G, n, Output:Boolean)

```
core=v, s.t. DoFE(v, G)≤DoFE(v', G), ∀v' ∈ V
return (DoFE(core, G) ≤ n)
```

function **DoFE**(Input: G, Vertex v, Output: Int)

```
∀e = (v_1, v_2) ∈ E, compute
    distance(v, v_1) + 1, denoted by dist(v, e)
return dist(v, e) s.t. dist(v, e) ≥ dist(v, e'), ∀e' ∈ E
```

Figure 14: Determining whether a Query is PWOC with Directed Hop Guarantee.

This algorithm assumes an undirected n-hop guarantee and the input query $G = \{V, E\}$ is an undirected graph. The optimizations for high-degree vertexes are turned on.

function **IsPWOC**(Input: G, n, Output: Boolean)

```
mark every vertex in G that is typed
unmark every vertex G that is typed high degree
core=v, s.t. DoFE(v, G)≤DoFE(v', G), ∀v' ∈ V
return (DoFE(core, G) ≤ n)
```

function **DoFE**(Input: G, Vertex v, Output: Int)

```
∀e = (v_1, v_2) ∈ E, compute
    min(Distance(v, v_1), Distance(v, v_2)) + 1,
    denoted by dist(v, e)
return dist(v, e) s.t. dist(v, e) ≥ dist(v, e'), ∀e' ∈ E
```

function **Distance**(Input: G, Vertexes v, w, Output: Int)

```
if v is unmarked
    if w is v's neighbor in G
        return 1
    else
        return infinity
else
    return the length of shortest path between v and w,
    of which all vertexes must be marked except w
```

Figure 15: Determining whether a Query is PWOC with Optimizations for High-Degree Vertexes.