Implementing Iterative Algorithms with SPARQL

Robert W. Techentin, Barry K. Gilbert
Mayo Clinic
Rochester, MN
{techentin.robert, gilbert.barry}@mayo.edu

Adam Lugowski, Kevin Deweese, John Gilbert
UC Santa Barbara
Santa Barbara, CA
{alugowski,kdeweese, gilbert}@cs.ucsb.edu

Eric Dull, Mike Hinchey, Steven P. Reinhardt
YarcData LLC
Pleasanton, CA
{edull,mhinchey,spr} @yarcdata.com

ABSTRACT

The SPARQL declarative query language includes innovative capabilities to match subgraph patterns within a semantic graph database, providing a powerful base upon which to implement complex graph algorithms for very large data. Iterative algorithms are useful in a wide variety of domains, in particular in the data-mining and machine-learning domains relevant to graph analytics. In this paper we describe a general mechanism for implementing iterative algorithms via SPARQL queries, illustrate that mechanism with implementation of three algorithms (peer-pressure clustering, graph diffusion, and label propagation) that are valuable for graph analytics, and observe the strengths and weaknesses of this approach. We find that writing iterative algorithms in this style is straightforward to implement, with scalability to very large data and good performance.

Keywords

graph analysis, SPARQL, data mining, iterative algorithms, clustering, query languages, performance

1. OVERVIEW

The SPARQL declarative query language [7] implements innovative capabilities to match subgraph patterns within a semantic graph database, providing a powerful base upon which to implement complex graph algorithms for very large semantic (or heterogeneous) data. SPARQL has major advantages for practical problem-solving, including its built-in support for semantic graph querying, its status as an emerging standard from the W3C along with its companion Resource Description Framework (RDF) [12] data format, and its implementation by numerous providers of both databases and tools, including Jena [1], Sesame [10], AllegroGraph [3], TopBraid Composer [18], and Urika [19]. The use of SPARQL is growing, so understanding its current capabilities and limitations is valuable, so it can be used to address the widest practical range of graph-analytic problems.

(c) 2014, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference (March 28, 2014, Athens, Greece) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

Iterative algorithms are useful in a wide variety of domains related to graph analytics, esp. data mining and machine learning, so having such algorithms readily implementable in SPARQL extends the range of practical algorithms considerably. We present one approach for implementing iterative algorithms in SPARQL, consisting of a) a set of initial queries that establishes a baseline state, b) a set of iterative queries that updates the state (typically via SPARQL Update constructs) and calculates the current value of convergence criteria, and c) a set of final queries that creates final results and cleans up intermediate state.

We illustrate this method via the implementation of three algorithms that calculate per-vertex metrics that depend on the structure of the graph. *Peer-pressure clustering* [15] groups vertices into clusters based on the cluster to which most of a vertex's neighbors belong. *Graph diffusion* [5] calculates the diffusion of an effect from seeded nodes throughout the graph, identifying both vertices that are likely to be related as well as pathways that contribute to the relationship. *Label propagation* [11] propagates known outcomes from a set of labeled data through a set of unlabeled data, tagging vertices with their likely outcomes based on the information latent in the graph.

We find that writing iterative algorithms in this style is straightforward to implement, with scalability to very large data and good performance. Though there are algorithms for which the iterative queries are so simple that the overhead of executing any query may be a performance issue, initially implementing such algorithms in this style delivers correct answers quickly, with an optimized implementation possible via other means if needed.

While there are other approaches to this problem, notably the work on recursive database queries with Datalog [16], our focus is on SPARQL because of its intended audience of subject-matter experts, not professional programmers.

2. THE SPARQL 1.1 LANGUAGE

SPARQL is a query language for semantic-graph databases containing data represented in the Resource Description Framework (RDF) [12] , with its name being a recursive acronym for SPARQL Protocol and RDF Query Language. It comes from the semantic web community and is a recommendation of the World Wide Web Consortium [4]. The primary goal for RDF was to make web pages machine-readable, and the goal for SPARQL was to enable higher-level querying of the semantic web. The resulting capabilities proved to be valuable for graphs that did not necessarily originate as web pages; *i.e.*, queries on highly heterogeneous

and richly interconnected data, data that reflected the Open World Assumption [5] that one's set of data is never complete and so tools must be built expecting to easily incorporate new data and new types of data. Readers who use SQL [6] will find many SPARQL constructs familiar.

RDF defines data in terms of *triples* consisting of a subject, a predicate or relationship, and an object. For example, the triple "Ruth works-for Mayo-Clinic" has "Ruth" as the subject, "works-for" as the predicate, and "Mayo-Clinic" as the object. Well-defined RDF data will use Universal Resource Identifiers (URIs, [9]) for subjects, predicates, and most objects. An RDF graph is a collection of these triples.

An example of SPARQL graph matching comes from the Lehigh University Benchmark (LUBM) query #2 [6]: (SPARQL keywords are shown in upper case for clarity)

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-
       rdf-syntax-ns#>
2  PREFIX ub: <http://www.lehigh.edu/~zhp2
       /2004/0401/univ-bench.owl#>
3  SELECT ?student ?faculty ?course
4  WHERE {
5    ?student rdf:type ub:Student .
6    ?faculty rdf:type ub:Faculty .
7    ?course rdf:type ub:Course .
8    ?student ub:advisor ?faculty .
9    ?faculty ub:teacherOf ?course .
10   ?student ub:takesCourse ?course
11 }
```

SPARQL variables are denoted by an initial ? or $ character, *e.g*, `?student` in the example above. The statements within the WHERE clause, known as a *basic graph pattern*, can be interpreted as "find all triples (`?student`, `?faculty`, `?course`) where (lines 5-7) `?student`, `?faculty`, and `?course` are of the corresponding types, and there exists (8) an edge of type `ub:advisor` from the `?student` vertex to the `?faculty` vertex, (5) an edge of type `ub:teacherOf` from the `?faculty` vertex to the `?course` vertex, and (6) an edge of type `ub:takesCourse` from the `?student` vertex to the `?course` vertex." In natural language, the query can be stated "find triples of Student, Faculty, and Course where the student takes a course taught by her advisor."

Once a graph pattern has been matched, the intermediate solution can be further processed or combined with other intermediate solutions. *E.g.*, LUBM query #2 could be modified as follows. The inner SELECT query (lines 4-14 below) matches the basic graph pattern in the WHERE, groups those results first by `?faculty` and then within a single `?faculty` value by `?student`; then those groups are aggregated by selecting the (unique, per group) distinct value of `?faculty` and `?student` and by COUNTing the instances of `?course` per group, and then keeping only the results that have a `?courseCount` greater than 2. The outer query takes the results of the inner query, groups them by `?faculty` and COUNTs the number of students (per faculty) who have taken more than one course from their advisor. The full result is faculty members who have advisees who have taken more than one course from them, sorted in descending order of the count of such advisees per faculty member.

```
1  SELECT ?faculty
2      (COUNT(?student) AS ?studentCount)
3  WHERE {
4    SELECT ?faculty ?student
5        (COUNT(?course) AS ?courseCount)
6    WHERE {
7      ?student rdf:type ub:Student .
8      ?faculty rdf:type ub:Faculty .
9      ?course rdf:type ub:Course .
10     ?student ub:advisor ?faculty .
11     ?faculty ub:teacherOf ?course .
12     ?student ub:takesCourse ?course
13   } GROUP BY ?faculty ?student
14     HAVING (?courseCount > 1)
15 } GROUP BY ?faculty
16   ORDER BY DESC(?studentCount)
```

We note other SPARQL constructs, including *named graphs*, each of which segregates a set of triples from the main body (*default graph*) of the graph database, enabling its simple identification for a specific use. SPARQL also supports aggregate functions and mathematical operators sufficient for basic computations on query results. FILTER limits results by various function comparisons rather than graph pattern. MINUS, EXISTS, and NOT EXISTS offer different ways of reducing results by graph pattern matching. OPTIONAL is like SQL's LEFT JOIN, allowing bindings that may not be present for some results.

SPARQL 1.1 also includes a set of Update capabilities [4], including INSERT, which adds triples to the database based on matching within the existing data (like the WHERE basic graph patterns above); DELETE, as the converse; LOAD, which reads data from a disk file or other source into the graph database; and DROP, which deletes a graph.

The execution flow of a (sub)query is (1) the basic graph pattern in the WHERE, (2) any GROUP BY or HAVING and any aggregation or projection (*i.e.*, the operations after the SELECT keyword), and (3) any trailing solution modifiers, such as ORDER BY or LIMIT.

3. ITERATIVE ALGORITHMS IN SPARQL

SPARQL 1.1 as a language does not support iteration, so iterative algorithms will need a construct external to SPARQL to implement iteration. We have used JavaScript and Python scripts to implement the iterative code that calls SPARQL queries. The coarse structure is captured in the following pseudocode, where any of lines 1, 3, 4, or 6 could consist of multiple SPARQL queries.

```
1  establish initial state
2  do {
3    update state
4    measure convergence criteria
5  } while (convergence criteria not met)
6  establish final state and clean up
```

This structure reflects the assumptions that a) the intermediate state updated in line 3 is large enough that we want to retain it within the SPARQL endpoint for performance reasons, rather than transmitting it back to a client system for processing, and b) that the convergence criteria can be summarized to no more than a few scalars (*e.g.*, the number of vertices changing cluster assignment on this iteration, for

peer-pressure clustering). In the algorithmic implementations described here, we have chosen to place intermediate state (*e.g.*, cluster assignment) in a named graph for simplicity (*i.e.*, ease of finding inserted triples for debugging and other purposes) and performance (*i.e.*, eliminating a set of intermediate values via DROP of a named graph is fast compared with finding all the appropriate triples intermingled with other data), but placing the intermediate state in the default graph may be appropriate in some circumstances. Another degree of freedom for the algorithm developer is whether to preserve all intermediate named graphs until the algorithm completes (at the cost of more memory) or to delete intermediate named graphs just after their last use (at the cost of debuggability). Preserving intermediate graphs requires emitting a query where the intermediate-graph name changes from iteration to iteration.

A similar choice for the algorithm developer is whether to place the final results in the default graph or a named graph. Given the extreme flexibility of RDF and SPARQL, an iterative algorithm could be called with different parameters, for example propagating labels through vertices of types A-C on one call and types B-G on a later call; the ability to name a graph with this information simplifies the other URIs within that data (*e.g.*, avoiding the need to express in the predicate (in this example) the specific set of vertex types considered).

A programming note is that nothing in the SPARQL endpoint precludes multiple instances of an iterative algorithm running simultaneously, each sending queries to the endpoint, so some means of avoiding collisions in intermediate graph names will be warranted for general use.

4. PEER-PRESSURE CLUSTERING

Peer-pressure clustering belongs to the class of algorithms that are effective by calculating simply on very large data.

4.1 The Algorithm

Peer-pressure clustering takes as its input a set of edges, each between a pair from a set of vertices, and calculates each vertex's assignment to a cluster. The number of clusters to be found need not be specified. For heterogeneous graphs, even for clustering vertices of homogeneous type, creating links between the vertices is an initial step whose definition is problem-dependent; see Section 4.6.1 below for details.

Reprising the structure from the figure above, peer-pressure clustering can be expressed via the following pseudocode.

```
1 assign each vertex to an initial cluster
2 do {
3   (re-)assign each vertex to the cluster
        to which a plurality of its
        neighbors belong
4   count the number of vertices that
        changed cluster in the prior step
5 } while (enough vertices changed or other
    criteria)
```

In our implementation the initial assignment is to a cluster with the same name as the vertex.

4.2 Relevant Use Cases

Clustering can be useful to understand the group structure of a set of homogeneous vertices. Use cases include the spread of influence in online social networks [8, 2].

4.3 Implementation via SPARQL Queries

We chose to place intermediate assignments in named graphs whose names have a common quasi-random seed, "`xjz`" in the examples below, to avoid collisions.

The first initial query assigns each vertex to a default cluster (named by the vertex name).

```
1 DROP GRAPH <urn:ga/g/xjz0>
2 CREATE GRAPH <urn:ga/g/xjz0>
3 INSERT {
4     GRAPH <urn:ga/g/xjz0>
5         {?s <urn:ga/p/inCluster> ?s }
6 } WHERE {
7    SELECT DISTINCT ?s
8    WHERE {
9       ?s <urn:ga/p/hasLink> ?o .
10   }
11 }
```

Line 1 DROPs (deletes from the database) any existing graph of the same name and thus any triples in such a graph. Line 2 CREATEs a new (empty) graph of the same name, which is not strictly necessary before the INSERT but can aid in debugging. The SELECT clause on lines 7-10 finds all vertices in the default graph that are the subject of a `hasLink` predicate and, for each unique such vertex, then on lines 3-5 INSERTs into the named graph a new triple of the same subject, the `inCluster` predicate, and the subject (as the cluster assignment). (We omit the text of a trivial initial query that counts the number of vertices to be clustered.)

The update query works as follows. Lines 13-20, for each vertex, return the vertex, the cluster assignments of its neighbors, and the per-cluster count of neighbor vertices. Lines 10-22 calculate the maximum cardinality of the clusters of the neighbors of each vertex. Lines 23-31 calculate the cluster assignment of that maximum-cardinality cluster. (SPARQL lacks a construct that returns the maximum value of one intermediate result and the corresponding element of another intermediate result.) Lines 8-30 join the maximum cardinality with the cluster name and also, in the case of a tie in maximum cardinality, break any tie by SAMPLEing a cluster assignment for each cluster. Lines 3-6 INSERT the new cluster-assignment triples into the named graph. (For all graph names, the non-SPARQL "*`[i]`*" and "*`[i+1]`*" syntax denotes that the appropriate iteration count is placed into the string by the code that creates the SPARQL query.)

```
1  DROP GRAPH <urn:ga/g/xjz[i+1]>
2  CREATE GRAPH <urn:ga/g/xjz[i+1]>
3  INSERT
4  {
5    GRAPH <urn:ga/g/xjz[i+1]>
6        { ?s <urn:ga/p/inCluster> ?clus3 }
7  } WHERE {
8    { SELECT ?s (SAMPLE(?clus) AS ?clus3)
9    WHERE {
10     { SELECT ?s
11         (MAX(?clusCt) AS ?maxClusCt)
12     WHERE  {
13       SELECT ?s ?clus
14           (COUNT(?clus) AS ?clusCt)
15       WHERE
16       {
17         ?s <urn:ga/p/hasLink> ?o .
18         GRAPH <urn:ga/g/xjz[i]>
19           {?o <urn:ga/p/inCluster> ?clus}
20       } GROUP BY ?s ?clus
21     } GROUP BY ?s
22     }
23     { SELECT ?s ?clus
24           (COUNT(?clus) AS ?clusCt)
25       WHERE
26       {
27         ?s <urn:ga/p/hasLink> ?o .
28         GRAPH <urn:ga/g/xjz[i]>
29           {?o <urn:ga/p/inCluster> ?clus}
30       } GROUP BY ?s ?clus
31     } FILTER (?clusCt = ?maxClusCt)
32     } GROUP BY ?s
33   }
34 }
```

The second query executed in each iteration (below) counts the number of vertices that changed cluster assignment in the just-completed iteration.

```
1 SELECT (COUNT(?oNew) as ?vccCt)
2 WHERE {
3   GRAPH <urn:ga/g/xjzi>
4     {?s <urn:ga/p/inCluster> ?oOld}
5   GRAPH <urn:ga/g/xjzi+1>
6     {?s <urn:ga/p/inCluster> ?oNew}
7   FILTER (?oOld != ?oNew)
8 }
```

The JavaScript code that constructs the queries and calls the SPARQL endpoint is straightforward and hence omitted.

4.4 Validation

We initially validated the implementation with synthetic data. The first phase of this was with predictable clustering characteristics, generated with the number of clusters set to $\log(n)^{1.5}$ where n is the number of vertices. The generator then considers all edge pairs and adds inter cluster edges with probability X (0.02 in this case) and intracluster edges with probability Y (0.1). This data contained 100,000 vertices and 15,736,484 triples.

The second phase of validation with synthetic data was block two-level Erdős–Rényi (BTER) data created by the MATLAB generator by Pinar et al [14], whose output we converted into RDF. The parameters we used, in addition to power-law degree distribution, were $\gamma = 2$, maxdegree = 100, $\rho_{init} = 0.99$, and $\rho_{decay} = 0.8$. This data contained 1,643,915 vertices and 7,322,102 triples.

4.5 Performance

For the initial set of synthetic data (100,000 vertices and 15.7M edges), on a 64-processor, 2TB Urika appliance, peer-pressure clustering converged after 5 iterations, consuming 200.2 seconds in total. For the BTER synthetic data (1.6M vertices and 7.3M edges) it executed for 3h:09m, though it did not converge after 20 iterations, which was the maximum iteration setting. We also tried to apply the algorithm to the Smackdown data created by Mayo Clinic, both small portions and the full 2G (2 billion triples) dataset, where we encountered the quadratic issue described in the following section. We had wondered whether per-query overhead might be a performance issue, but with the overhead far below 1 second, it proved not to be an issue in practice.

4.6 Issues Encountered

4.6.1 *Creating links for clustering*

Peer-pressure clustering uses predicates of a given type (`hasLink` in our implementation) as the edges to consider, which can be viewed as similarity links between the vertices. For heterogeneous data, the data must typically be prepared by deciding the similarity criteria, and for vertex pairs which pass the criteria or threshold, creating the edge. We experimented with different similarity functions, mirroring what subject-matter experts may do in practice, tweaking the similarity function until the resulting clusters are useful in the context of the subject matter.

If this approach is used to calculate the edges, the simple approach of comparing all vertices to all other vertices is difficult to scale to large numbers of vertices, as the $O(n^2)$ cost of this step becomes prohibitively time-consuming for databases containing O(100M) or more vertices. The clustering algorithm doesn't require all similar vertices to have similarity links, but can work with a more sparsely connected graph, and a sparse pre-processing step would be appropriate for large-scale use. Note that this cost is in the pre-processing step, and that limiting the number of edges created would keep the core peer-pressure clustering algorithm relevant for very large data.

4.6.2 *SPARQL constructs*

Careful readers may note that the inner SELECTs at lines 13-21 and 23-30 in the iterative update query are identical. While developing a complex nested query like this, needing to keep the same code in two spots identical is cumbersome. SPARQL 1.1 contains no good mechanism to define this code once and reuse it, like a function in a procedural language. The SQL WITH clause defines by name such a code block that can be executed wherever its results are needed.

The second and third queries above both have minor changes from one instance to the next (*e.g.*, substituting "`xjz2`", and "`xjz3`" into the graph name). While these are not hard to cope with in JavaScript code that creates the queries, it does mean that the query is literally different each time it is executed, and hence the SPARQL endpoint will have to reinterpret and re-optimize the query each time, which could at some point become time-consuming. SQL?s *placeholder* capability enables the passing of a value (of a given type) at

execution time that is inserted at the placeholder's position in the query, avoiding reinterpretation.

5. GRAPH DIFFUSION

Graph diffusion is an algorithm that models natural transport phenomena on the connectivity of the graph, much like random walk approaches, but simultaneously moving across all possible edges. Diffusion can be used to characterize semantic data in several ways. It can be used to compute neighborhoods of "close" connectivity or find nodes with similar features. Some applications are models for cascading behaviors such as social network analysis, virus propagation, parallel load balancing, and chemical compound classification [5] [13] [17].

5.1 The Algorithm

The graph diffusion algorithm propagates values (typically numeric scores) from specific initial vertices, through connecting edges to neighboring vertices, and by iteration, to the rest of the graph. Each vertex accumulates values as the expanding wavefront propagates through it. Semantic graphs, with named edges, can have a propagation weight assigned to each edge type, increasing or decreasing the diffusion values. In general matrix notation, graph diffusion can be characterized as an iterative update process according to the equation, $N_i^{t+1} = \sum_j E_{ij} W_{ij} N_j^t$, where N is a node (vertex), E is the binary adjacency matrix (representing edges between nodes), and W contains the edge weights associated with diffusion. In this formulation, E and W are static and can be combined. The diffusion algorithm can run a fixed number of iterations, or (with edge weights < 1.0) can continue until a steady state is reached. There are many algorithm parameters which can be adjusted, including initial conditions, treatment of edges as directed or undirected, and computation of aggregate scores.

A simple implementation of graph diffusion is characterized by this pseudocode.

```
1 assign edge weights by type
2 seed initial diffusion value(s)
3 do {
4     for each vertex with a value {
5       save value as accumulated score
6       propagate value to neighbors
7     }
8 } until completion criteria met
```

5.2 Relevant Use Cases

One potential use case for diffusion in healthcare is finding patients with similar clinical features. For example, patients are admitted into a hospital for a variety of different clinical conditions; however, once admitted, apparently dissimilar patients develop common presentations of disease. We applied diffusion to a semantic graph of two years of hospital records for 114,943 patient stays. Diffusion seed values were attached to sub-populations of patients with known conditions of interest. Diffusion values propagated over 74 different edge types (representing demographic, clinical, nursing, and lab measurements) from the initial patients to all others in the dataset. The resulting diffusion values represent each patient's similarity to the initial patient sub-population.

5.3 Implementation via SPARQL Queries

Edge (predicate) weights and diffusion values were stored in named graphs, separating them from each other and the patient data in the default graph. Propagation values were stored in numbered named graphs (*e.g.*, `iter_0`). Propagation values were accumulated for each patient at the end of the process, avoiding per-iteration updates to vertex counters. The SPARQL code for the first diffusion iteration uses values in `iter_0` to create values in named graph `iter_1` .

```
1  PREFIX diff: <urn:diffusion/>
2  DROP GRAPH diff:iter_[i+1];
3  CREATE GRAPH diff:iter_[i+1];
4  INSERT {
5    GRAPH diff:iter_[i+1]
6      {?vertex diff:cntr ?value .}
7  } WHERE {
8    SELECT ?vertex (SUM(?edgeVal) AS ?value)
9    WHERE {
10     GRAPH diff:iter_[i]
11       {?otherVertex diff:cntr ?otherCntr}
12     GRAPH diff:weights
13       {?edge diff:weight ?weight .}
14     { {?otherVertex ?edge ?vertex .}
15       UNION
16       {?vertex ?edge ?otherVertex .} }
17     BIND(?otherCntr*?weight AS ?edgeVal)
18   }  GROUP BY ?vertex
19 }
```

The graph pattern in lines 10-11 matches diffusion values from the previous iteration; lines 12-13 identify edge weights in a separate named graph; and the UNION operation at line 15 matches both incoming and outgoing edges of the vertex, making diffusion bidirectional. The BIND on line 17 computes the edge-weighted diffusion value, and the subquery on line 9 aggregates the values. For this implementation, both incoming and outgoing edges of the same type are weighted equally, but a straightforward extension would provide different weights based on edge direction.

All iterations use identical SPARQL code, save for incrementing the graph number on lines 2, 3, 5, and 10. Note that due to the nature of intermediate solution sets and grouping by `?vertex`, this code computes a new diffusion value for a vertex from its neighbors, as opposed to propagating a vertex's value to its neighbors as shown in the pseudocode.

After diffusion is complete, the per-patient similarity scores are found in the final iteration's named graph, `iter_4`. More generally, aggregate diffusion scores are computed by summing vertex counters in all iteration-named graphs.

5.4 Validation

This implementation of graph diffusion, applied to this in-hospital patient care dataset, generates a score ranking all patients' similarity to the seed patient. The similarity score is computed over all 74 dimensions (edge types) in the semantic model. Inspection of the values for "similar" and "dissimilar" patients, one dimension at a time, reveals that the most similar patients do, in fact, have characteristics similar to the seed patient.

5.5 Performance

We ran four iterations of the diffusion algorithm on a semantic dataset of 89 million vertices and 1.8 billion edges

on a 64 processor Urika appliance with 2 TB of main memory. The first two iterations, propagating from the initial patient URIs to all associated values recorded for those patients, took 202 and 207 seconds, respectively. The third and fourth iterations, propagating back to all other patient URIs, took 1,302 seconds and 1,124 seconds, respectively.

5.6 Issues Encountered

For this RDF dataset, extracted from tabular data, the basic structure and relationships between patient URIs, measurement events, and associated data values were well known. This structure allowed us to determine that four iterations of diffusion were sufficient to propagate from the initial patient URI to all other patient URIs. In the more general case, a scoring query would be needed after every UPDATE iteration to determine if the diffusion had completed.

6. LABEL PROPAGATION

Label propagation [11] is a clustering algorithm similar to peer-pressure clustering whose purpose is to find clusters of vertices where the clustering is based on the edges linking to neighboring vertices.

6.1 The Algorithm

Clustering via label propagation takes as its input a set of edges, each between a pair of vertices, and calculates each vertex's assignment to a cluster based on its neighbors.

Reprising the structure from the figure above, label propagation can be expressed via the following pseudocode:

```
assign each vertex to a distinct cluster
  (cluster label might be some integer
      primary key of the vertex)
do {
  re-assign each vertex to a cluster based
       on neighbors in the previous step
    - choose cluster to which a plurality
        of first-degree neighbors belong
    - optional self-voting
    - tie-breaking rule: sort order of
        label
} while not stopping conditions
```

6.2 Implementation via SPARQL Queries

In this implementation, the intermediate assignments are placed in the same (default) graph, but with a distinct predicate name. This makes little difference functionally, and the SPARQL queries are easier to only a minor degree because the syntax for specifying multiple different graphs is more verbose.

The initial cluster label is denoted by the `grouping0` predicate, and assigned the integer primary key of the vertex. The predicate `pkey` is specific to the source data model.

The `p:similar` predicate is also specific to the data model; it represents the network linking the vertices (`?entity`), analogous to `hasLink` in the peer-pressure clustering section.

```
INSERT {
  ?entity lprop:grouping0 ?initial_group
} WHERE {
  ?entity p:similar ?x .
  ?entity p:pkey ?initial_group .
} ;
```

In each iteration, the grouping predicate is incremented to keep them distinct, `lprop:grouping0` in the `WHERE` to look at the previous iteration, and `INSERT`ing into `lprop:grouping1`.

The query consists of the `INSERT` and a `WHERE` with nested sub-queries. Sub-queries are joined on bindings of the same names, in the same way as simple *basic graph patterns*. (The query is shown below broken into multiple code listings.)

```
INSERT { ?entity lprop:grouping1 ?group .
} WHERE {
```

The sub-query selects the entity and group. The `GROUP BY` at the end is on `?entity` because the query needs one solution for each `entity`. The `MIN` of the `group` is chosen to break ties in popularity. That is, if multiple groups have the same popularity, the lowest `pkey` value wins. This is an arbitrary rule, but consistent.

```
  {
    SELECT ?entity
      ( MIN( ?groupx ) AS ?group )
    WHERE {
```

The first sub-sub-query measures the popularity of the labels of each entity's first-degree neighbors. The `GROUP BY` includes `?entity` to find results per entity; and `?group` so it can `COUNT` the neighbors (`?first`).

In addition, we've chosen to use self-voting, adding 1 to the `popularity` score for the entity's group (`?self`). Note that this query alone might have multiple binding-sets for an entity, one for each group, and that multiple groups might have the same popularity.

```
      {
        SELECT ?entity ?groupx
          (( COUNT( ?first ) +
             IF( ?self = ?groupx, 1, 0))
           AS ?popularity )
        WHERE {
          ?entity p:similar ?first .
          ?entity lprop:grouping0 ?self .
          ?first lprop:grouping0 ?groupx .
        }  GROUP BY ?entity ?self ?groupx
      }
```

The next sub-sub-query contains another sub-query which is the same popularity query as above, with the binding names changed so the outer query can use the correct names. The outer query selects the `MAX popularity` for each entity. However, SPARQL does not have a direct mechanism to select the group(s) that have that popularity. This is the reason for duplicating the popularity query. This query below and the one above are joined on `?entity` and `?popularity`, which leaves the `?groupx` in the results.

(Closing out the query is the `GROUP BY` explained above.)

```
      {
        SELECT ?entity
          ( MAX( ?popularityx ) AS ?
              popularity )
        WHERE {
          {
            SELECT ?entity ?groupx
              ( ( COUNT( ?first ) +
                  IF( ?self = ?groupx, 1,
                      0 ) )
```

```
9                 AS ?popularityx )
10              WHERE {
11                ?entity p:similar ?first .
12                ?entity lprop:grouping0 \\?
                      self .
13                ?first lprop:grouping0 \\?
                      groupx .
14              }
15                GROUP BY ?entity ?self ?
                      groupx
16            }
17          }  GROUP BY ?entity
18        }
19      }  GROUP BY ?entity
20    }
21  }
```

A variation would be to count second-degree (or greater) neighbors instead of (or besides) first-degree neighbors.

There are a number of conditions for stopping the iteration. As with the peer-pressure algorithm above, the percent or number of vertices changed, and simply the maximum number of iterations are considered. There is also a property to the algorithm such that one iteration can make numerous changes, but the next iteration will reverse them, resulting in oscillation that only ends with the max number of iterations. Our implementation detects this by counting the differences between each iteration (below `grouping0` and `grouping1`), then also the previous iteration (which would be `grouping0` and `grouping2`). If either of these `group_diff_count`s are `0`, the algorithm halts.

```
1 SELECT ( COUNT( ?entity ) AS ?
      group_diff_count )
2 WHERE {
3   ?entity lprop:grouping0 ?label .
4   ?entity lprop:grouping1 ?next .
5   FILTER( ?label != ?next )
6 }
```

6.3 Validation

We generated synthetic, semi-random data, with some number of groups expected, and some links fully random, not belonging to the expected groups.

When run on real data, reports were generated by SPARQL queries (not shown) to show the averages and modes of various attributes for each group, from which we could see that the groups did have distinct sets of attributes.

6.4 Performance

Generation of synthetic data was done in Python, but all other processing of big data was performed by SPARQL within the Urika database. A 64-processor, 2TB Urika appliance was used for running label propagation.

Using a semi-random network of 1.8M vertices and 8M edges, the process stopped after 20 iterations, in 16 minutes, resulting in 48 groups. With 3.6M vertices and 16M edges, the process stopped after 20 iterations, in 26 minutes, resulting in 49 groups.

Calculating the same "popularity" sub-query twice in each iteration is redundant and expensive. The alternative is to first insert the results of that once, then use the results twice. The trade-off is the cost of inserting a very large result set, which may take more time than building it twice.

6.5 Issues Encountered

Long SPARQL queries with repeated (and slightly modified) sections are difficult to maintain. The program driving the iteration and calls to the SPARQL endpoint was implemented in Python. A simple module was written to templatize SPARQL text to make portions reusable within a query and across queries, while avoiding the need to mix Python and SPARQL code.

Manually debugging any algorithm like this can be challenging, especially with large data. Since the data never leaves the database during processing, the usual practices for debugging software (such as a Python script) do not apply. In addition, the algorithm operates differently from how many scripts are written: breadth first rather than depth first. Using a smaller dataset makes the manual inspection approachable, but this changes the results. Also, to understand the results of each iteration requires more SPARQL queries to inspect or count the results.

Running this type of algorithm can take many hours. If the process stops early because of error, or because the user needs it stopped, starting over from the beginning wastes a lot of time. Inserting metadata into the database itself as each step is completed, allows for the process to be restarted, continuing where it was stopped.

7. DISCUSSION

SPARQL as a query language possesses several positive features that lend to rapid prototyping of graph analysis workflows. These features include easy graph preparation, the expressibility of algorithms in the language, and the accessibility of the language to individuals who possess relevant domain expertise but may not possess sufficient knowledge of lower-level graph languages to effectively use them.

Building graphs from real world data on which to apply these algorithms tends to require graph preparation. This graph preparation step is expressible in SPARQL. This step serves to focus analyst and developer attention on questions about which types of data and aspects of that data are most relevant to the domain-specific problem being analyzed. This focusing is an important part of the graph workflow prototyping process as it forces critical thought to the practicalities of the data representation and associated algorithms.

The three algorithms have different execution speeds, measured in edges/second, reflecting the different natures of the algorithms and the newness of these results. In future work, we will explore how much of these speed differences are inherent to the algorithms and how much could be overcome by query changes or query-engine optimizations.

Iterative algorithms are a valuable component of graph analysis workflows. These algorithms produce analytically relevant results, as shown in the rest of the paper, and they are expressible in SPARQL.

Developing, prototyping, and evaluating graph analysis workflows tend to be laborious, human-intensive processes. This development and prototyping requires expertise in graph analysis and the specific domain under analysis. Practitioners with these experiences tend to have less knowledge and familiarity with lower-level graph processing frameworks and languages. SPARQL as a high-level graph query language is

more approachable to these practitioners and requires less time to learn than C++ or another lower-level language. This smaller ramp-up time translates to a faster idea-to-functioning-workflow when using SPARQL than other graph-workflow development languages.

8. SUMMARY

We present a method of mapping iterative algorithms on to a combination of SPARQL queries and code in a procedural language (Python and JavaScript, in our examples) that calls the SPARQL queries. We find that writing iterative algorithms in this style is straightforward to implement, with scalability to very large data and good performance. Though there are algorithms for which this approach may be problematic (*e.g.*, when iterative queries are so simple that query overhead is a performance issue, or the intermediate state between iterations is prohibitively large), initially implementing such algorithms in this style delivers correct answers quickly, with an optimized implementation possible via other means if needed.

9. ACKNOWLEDGMENTS

The authors from UC Santa Barbara were supported in part by DOE Office of Science contract DE-AC02-05-CH-11231, NSF grant CNS-0709385, a contract from Intel Corp., and a gift from Microsoft Corp.

10. REFERENCES

[1] Apache Jena Project. Java framework for building linked data applications., 2011-2013. http://jena.apache.org/.

[2] M. Cha, A. Mislove, and K. P. Gummadi. A measurement-driven analysis of information propagation in the Flickr social network. In *Proceedings of the 18th International Conference on World Wide Web*, pages 721–730. ACM, 2009.

[3] Franz, Inc. AllegroGraph ontology modeling capabilities, 2013. http://www.franz.com/agraph/.

[4] P. Gearson, A. Passant, and A. Polleres. SPARQL 1.1 update (W3C recommendation 21 march 2013). Technical report, World Wide Web Consortium, 2013. http://www.w3.org/TR/sparql11-update/.

[5] D. Gruhl, R. Guha, D. Liben-nowell, and A. Tomkins. Information diffusion through blogspace. In *In WWW '04*, pages 491–501. ACM Press, 2004.

[6] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3), 2005.

[7] S. Harris and A. Seaborne. SPARQL 1.1 query language (W3C recommendation 21 march 2013). Technical report, World Wide Web Consortium, 2013. http://www.w3.org/TR/sparql11-query/.

[8] P. Hui and S. Buchegger. Groupthink and peer pressure: Social influence in online social network groups. In *Proceedings of International Conference on Advances in Social Network Analysis and Mining (ASONAM)*. IEEE, 2009.

[9] Network Working Group. Uniform resource identifier (URI): Generic syntax (RFC 3986). Technical report, Internet Engineering Task Force, 2005. http://tools.ietf.org/html/rfc3986.

[10] OpenRDF Project Team. OpenRDF.org ... home of sesame, 1997-2012. http://www.openrdf.org/.

[11] U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3), March 2007.

[12] RDF Working Group. Resource Description Framework (RDF). Technical report, World Wide Web Consortium, 2004. http://www.w3.org/RDF/.

[13] S. Schamberger. On partitioning fem graphs using diffusion. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 2004.

[14] C. Seshadhri, T. G. Kolda, and A. Pinar. Community structure and scale-free collections of Erdős-Rényi graphs. *Physical Review E*, 85(5), May 2012.

[15] V. B. Shah. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. PhD thesis, University of California, Santa Barbara, 2007.

[16] A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. In *Proceedings of IEEE Conference on Very Large Databases*. IEEE, 2013.

[17] A. Smalter, J. Huan, Y. Jia, and G. Lushington. Gpd: A graph pattern diffusion kernel for accurate graph classification with applications in cheminformatics. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 7(2):197–207, 2010.

[18] TopQuadrant. TopBraid Composer modeling environment, 2013. http://www.topquadrant.com/products/TB_Composer.html.

[19] YarcData LLC, a Cray Company. Urika graph-analytic appliance, 2013. http://yarcdata.com/Products/.