RADAR: Radial Applications' Depiction Around Relations For Data-Centric Ecosystems

Panos Vassiliadis

Department of Computer Science, University of Ioannina
45110, Ioannina, Hellas
pvassil@cs.uoi.gr

***Abstract*—A data-centric ecosystem is the configuration of a central database along with all the applications that operate on top of it and are (a) dependent upon the database for performing their task appropriately and (b) closely related in terms of coding to the database. This paper presents first results towards addressing the issue of visualization of such a data-centric ecosystem in a reference "map". We map SQL queries and relations to a graph and present alternative algorithms for the visualization of the graph. First we start with a barycenter-based method that puts all queries in a line and all relations in a parallel line, in a way that closely put nodes are the ones that are most similar (within the same line) and closely related (across lines). Then, we move on to extend it with a variant that involves multiple lines for the queries. Finally, we present our main algorithm Concentric Radial, for placing queries and relations in multiple concentric circles. This arrangement allows the extensibility of the diagram and the exploitation of the canvas, the customization to alternative application needs, and, finally serves better the purpose of spatial memory.**

I. INTRODUCTION

Assume a database surrounded by a large variety of applications depending on it. For example, data entry or query forms are used by hundreds of users updating or querying information; complex workflows that operate in the enterprise frequently pose queries to the database; analysts working with pre-canned reports and OLAP tools pump data from the database; and so on and so forth. We refer to such an environment as a **data-centric ecosystem**, as it involves a large amount of software modules deployed around a database, and, at the same time, being unable to operate without the database operating properly. *We would like the administrators of the database, as well as the developers of all the software modules of the ecosystem to be able to have a visual overview of the ecosystem and to be able to understand the interdependencies between its parts. In simple words, the question can be expressed as: "how do we visualize a data-centric ecosystem?"*

The problem is both important and neglected by the database community. DeLine, Venolia and Rowan in a recent paper in ACM Queue [4] make a really solid case on the necessity of a reference "map" for the ecosystem. A survey over 400 Microsoft employees has distilled 3 main reasons where a diagrammatical representation is essential for the developers' tasks: code understanding, code design and refactoring, and ad-hoc meetings. Also, one can easily see how important the visual representation can be for handling database evolution and performing impact analysis. We refer the interested reader to [4] and [1] for a broader discussion of the importance of a visual representation of code to developers – with the reminder that although the discussion is mainly aimed to object-oriented code, the context is clearly relevant to application developers working over a data-centric ecosystem. In the field of object-oriented development (with large numbers of modules and classes) there exist several efforts for allowing the developer to navigate efficiently throughout the vastness of software modules. Early efforts include Treemaps [6] and SeeSoft [5] whereas recent follow-up efforts include Code Thumbnails [2], Code Canvas [4], and, Code Bubbles [1].

Despite these efforts, the data management community has not dealt with the issue so far. So, in this paper, we report on our preliminary findings of on-going research around the issue of visualizing a data-centric ecosystem. We map SQL queries and relations to a graph and try to exploit the idiosyncrasies of the ecosystem in order to graphically depict it as effectively as possible. The main requirements for the representation methods we have explored are:

- Clear separation of relations and queries; this has to do both with the nature of these components per se, and also with the fact that frequently, they are managed by different groups of people (developers and administrators)
- Spatial memory as the means to ease browsing, i.e., placing graph's nodes that have semantic relationships closely over the diagram
- Simple representation as the means to intuitively demonstrate the ecosystem's structure to the observer.

We have followed a fundamental approach and we have started with a simple representation technique, a barycenter-based method that exploits the fact that the ecosystem's graph is bipartite (edges connect only queries to relations) and depicts relations and queries in two parallel lines, with similar nodes placed closely on the diagram. To define similarity, we treat queries that hit the same relations as similar and reuse this definition for relations, respectively. Then, we extend this method by allowing multiple layers of queries surround the line of relations. This idea of surrounding the line of relations has lead us to employ a radial deployment of the graph's nodes over the canvas. The main contribution of this paper is the *concentric radial* representation based on the idea of depicting the ecosystem as a set of concentric circles, with the innermost circle keeping the database relations and subsequent circles hosting queries.

II. Background

In this paper, we model an ecosystem as a simple graph $G(V,E)$ with relations and queries as nodes and their interdependencies as edges. Specifically, the set of nodes G is defined as the union of the set of nodes V_R and the set of nodes V_Q, standing for relations and queries, respectively. An edge (q, r) connects a query q to a relation r if the query utilizes r in any way within its definition. Table I presents the reference notation used in the rest of our deliberations.

TABLE I
NOTATION FOR THIS PAPER

N_T	Number of relations of the ecosystem
N_Q	Number of queries of the ecosystem
relations(q)	the set of relations that are utilized by query q
queries(r)	the set of queries that access relation r

For our algorithms to work, we need to precompute the similarities between similar constructs. Assume two queries q_i and q_j. Then, the Jaccard similarity of q_i and q_j is computed as the following fraction:

$$JaccQ[q_i][q_j] = \frac{|relations(q_i) \cap relations(q_j)|}{|relations(q_i) \cup relations(q_j)|}$$

The Jaccard similarity for relations, $JaccT[r_i][r_j]$ is computed similarly by counting the fraction of the common queries of the two relations over the set of all the relations used by any of the two relations.

$$JaccT[r_i][r_j] = \frac{|queries(r_i) \cap queries(r_j)|}{|queries(r_i) \cup queries(r_j)|}$$

To compute the Jaccard similarities for relations we must scan all pairs of relations over all queries with a complexity $o(N_T \times N_T \times N_Q)$. Jaccard similarities for queries are computed respectively. All our algorithms that utilize node similarities require that these Jaccard similarities are precomputed.

It is interesting to highlight here that the exact semantics of the similarity function are orthogonal to the subsequent algorithms; if the user needs to depart from the abovementioned simple formulae for computing the similarity of two queries/relations (e.g., by taking into account the selections or aggregations performed by the queries) this does not affect the correctness of the forthcoming algorithms.

III. Layout algorithms in a two dimensional plane

In this section, we present the fundamental algorithms that we have employed in order to deploy the full ecosystem graph in a 2D plane. We start with the barycenter-based methods and then proceed to demonstrate our radial representation method. We will employ the TPC-H [7] benchmark as a reference example for the discussed algorithms.

A. Barycenter – based methods

The definition of the ecosystem's graph demonstrates that the graph is bipartite: all edges stem from the queries and target the relations (i.e., $E \subseteq V_Q \times V_R$). In principle, the goal of the visualization algorithm is to come up with a layered drawing of the ecosystem's graph [3]. The basic steps in such an approach are three: (i) assignment of nodes to layers, (ii) crossing reduction, and, (iii) horizontal coordinate assignment.

In our case, the layer assignment is simple, since there are two distinct and clear groups of nodes (queries and relations). However, the problem of reducing crossings between the two layers is NP-complete already [3] and basically reduces to the appropriate ordering of the nodes of the two layers (and not in the calculation of their exact x-coordinate). The most typically followed methods for minimizing crossings are the barycenter and median methods (or some of their variants).

A brief description of the two methods is due here. Assume that our problem consists only of two layers of nodes, L and L'. In order to determine the horizontal axis position of a node v of a layer L, the barycenter method tries to place it as close as possible to the "middle" of the line spanning the nodes in the other layer L' with which an edge with v exists. This is typically done, either by taking the median, or by computing the average (barycenter) value of their positions.

Overall, the problem can be split in two parts: (a) the placement of relations in a line, in a way that makes sense and, (b) the placement of queries in a parallel line in a way that also makes sense (of course, this could be done in the reverse order too). We discuss these sub-problems in the sequel.

1) The alignment of relations

As already mentioned, a typical barycenter-based method will align the relations of the ecosystem in a line; then, on the basis of this alignment, the method will align the queries of the ecosystem accordingly. In the context of our specific problem, we propose algorithm *AlignTablesAsL1* for the first task. The goal of algorithm *AlignTablesAsL1* is to organize the relations of the ecosystem in groups, in a way that the members of each group are closely related to each other. Remember that two relations are similar if they share a large amount of common queries that they sustain; thus by placing similar relations closely we aim to put the respective queries closely to them and avoid edges that span a large part of the drawing canvas (which, in return, hopefully minimizes the crossing with other edges too).

Algorithm *AlignTablesAsL1* (Fig. 1) assumes the existence of a drawing area defined by the values $[X_{low}, X_{upper}]$ and $[Y_{low}, Y_{upper}]$. The input to the algorithm is the graph $G(V,E)$, $V=V_R \cup V_Q$, $E \subseteq V_Q \times V_R$; the aforementioned limits of the 2D canvas, and an offset dY for the representation of the line of relations in it.

Algorithm *AlignTablesAsL1* proceeds as follows. While there exist relations not assigned to a position in the output list, the algorithm tries to find the two most similar relations out of the remaining ones, r_i and r_j, and once it finds them, it marks r_i as *maxI* and r_j as *maxJ*. Then, it places relation *maxI* to the sorted list's next free slot and adds right next to it all the relations whose similarity to *maxI* is larger than a user-defined threshold τ (which is given as input parameter to the algorithm). Practically, the sorted list contains sub-lists of relations, each with decreasing maximum similarity among its members. The node *maxJ* is in the first of these lists of relations. Then, the x,y coordinates of the relations are given in a simple manner: (a) all relations get the same y-coordinate as an offset from Y_{upper} (remember: it is a line parallel to the x-axis) and (b) each relation is placed in the next slot (computed as a dX step) in the x-axis.

```
Input: the graph $G(V,E)$, $V=V_R \cup V_Q$, $E \subseteq V_Q \times V_R$; the values $X_{low}$,
$X_{upper}$, $Y_{low}$, $Y_{upper}$, and, an offset $dY$ for the representation of the line
of relations
Output: a sorted list $L$ of these relations, with the x,y coordinates of
each relation computed
Begin
While there exist relations not visited yet{
        For every relation $r_i$ that has not been visited yet{
                For every other relation $r_j$
                        If $JaccT[r_i][r_j]$ is the current max
                          similarity, $maxI := r_i$ & $maxJ := r_j$ ;
                Place $maxI$ in the next slot in the sorted list $L$;
                For every other not visited relation $r_j$ {
                        If ($JaccT[r_i][r_j]$ > threshold $\tau$) place $r_j$
                            next to $maxI$;
                        Set $JaccT[r_i][r_j] = 0$;
//to avoid reconsidering it in the future
                }
        }
        If the last relation is not assigned, place it appropriately;
}//all relations are placed in the sorted list by now
Compute a $dX$ common to all relations, $dX = (X_{upper} - X_{low}) / (N_T + 1)$
For every relation $r_i$ assign coordinates x and y as follows:
        $r_i.x = X_{low} + dX * i$;  //relations in a line parallel to x-axis
        $r_i.y = Y_{upper} - dY$;              //dY is input to the algorithm
End.
```

Fig. 1. Algorithm *AlignTableAsL1*.

2) *The alignment of queries*

For every query q_i, we assign a constant y coordinate, in the same way that we have handled the relations. As far as the x coordinate is concerned, we do a barycenter computation: we add the x coordinates of all the relations accessed by the query and we divide this sum by the out-degree of the query. This places the query in the middle of the range defined by the relations that it accesses (see Fig. 3 for the TPC-H example).

Although this might appear adequate, it might place several queries very close to each other: so, we need to give some space to them. This is done by sorting them over their newly computed x coordinate and adding a dX interval to each of them. Practically, this places each query in its appropriate "slot" in the query line.

3) *An extension for two layers of queries*

A simple extension of the barycenter method involves employing two lines for the alignment of queries instead of one. These lines surround the line of relations, thus the name 'sandwich barycenter' deployment. The method is identical to the simple barycenter method with the following differences at the end of its execution: (a) the relation line is deployed in the middle of the vertical axis; (b) although the queries are 'sorted' as before, they are now aligned alternately to the upper and lower line that surround the middle relation line.

B. *Concentric Radial*

Apart from the barycenter methods that place nodes in parallel lines, we also explore the possibility of placing relations and queries in concentric circles. A *Naïve Radial* representation is possible by placing the queries and the relations in two homocentric circles, the inner *Relation Circle* for relations and the outer *Query Circle* for queries. However, this naïve radial algorithm does not take any actions towards the ordering of relations and queries, in order to minimize the edge crossings inside the diagram. Algorithm *Concentric Radial is* a significant extension of the Naïve Radial algorithm (and the other ones, too) with such considerations.

As the name implies, the algorithm Concentric Radial is based on the idea of placing queries (and relations) in a set of concentric circles. *Why concentric circles*? The fundamental motivation of pursuing this approach is to provide a simple and scalable presentation. The presentation is simple, as the discrimination of relations and queries is apparent. The scalability of the approach is basically due to the fact that the number of concentric circles can be extended at will. Moreover, the diagram can hide semantics in the concentric circles and can thus facilitate the specialized production of sub-diagrams in a personalized way. Here we list a few categories of semantics than can accompany each circle: (a) each circle can represent a class of queries in terms of structure; (b) each circle can represent the contents of a file and a group of closely placed, similarly colored circles can represent directory structure; (c) each circle may represent a different developer, or queries of different age; and so on and so forth.

The main goals of the *Concentric Radial* algorithm (Fig. 3) are:

(a) To place relations in a circle in the middle of the diagram and deploy query nodes around them;
(b) To distribute the query nodes at appropriate angles around the relations circle, such that each query is as close (in terms of angle) as possible to the relations it accesses;
(c) To de-clutter the diagram by placing query nodes of similar degree in homocentric cycles of increasing distance from the center (the larger the fan-out, the further the placement;
(d) To resolve problems of "conflicts" (i.e., placement of query nodes too close on the diagram) by slightly shifting conflicting query nodes.

We will need to establish the following parameters for our algorithm to work:

- A radius for the relations, ρ, that determines the size of the relations' circle. We must be cautious to pick a size that does not overload the circle with relation nodes and at the same time allows ample space for the concentric circles of the queries
- An angle θ that will determine the circular sector (which can be thought of as "pizza slice" slot) per relation in the circle; by default, we set this equally for all relations as the fraction of 360^o by the total number of relations

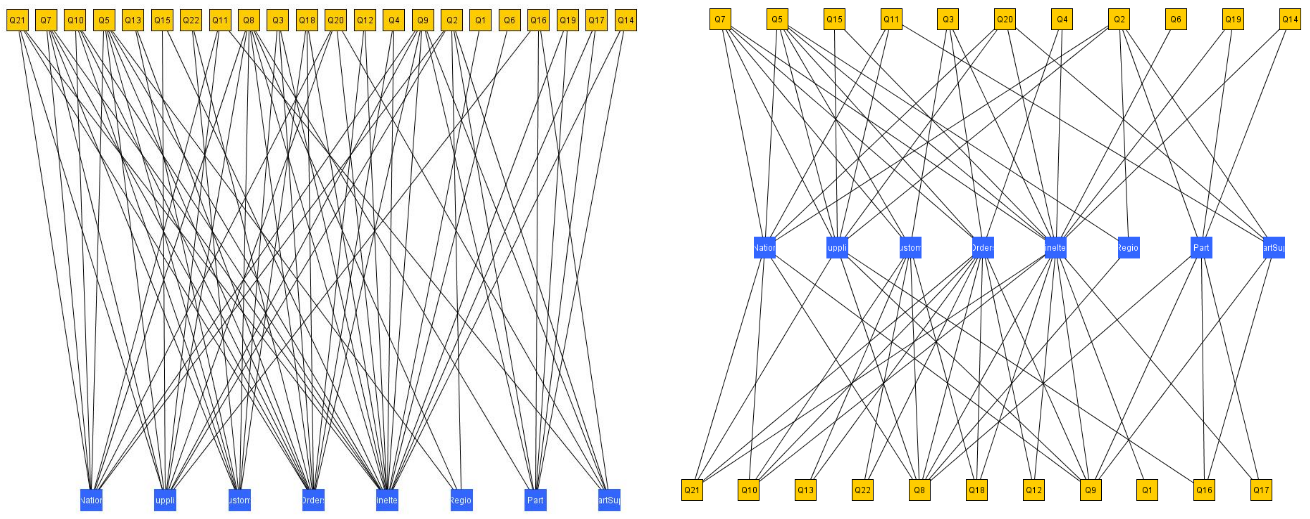


Fig. 2. The layout of the TPC-H ecosystem as the output of the Barycenter (left) and the Sandwich Barycenter (right) algorithms. Edges have been smoothened with organic edge routing by the yEd tool.

Input: A graph $G(V,E)$, $V=V_R \cup V_Q$, $E \subseteq V_Q \times V_R$; a default radius for the relations' circle, ρ and a center (x_0,y_0); the empty space radius between the concentric circles Δ_ρ; a small delta radius for conflicting queries $\Delta_\rho^{conflict}$

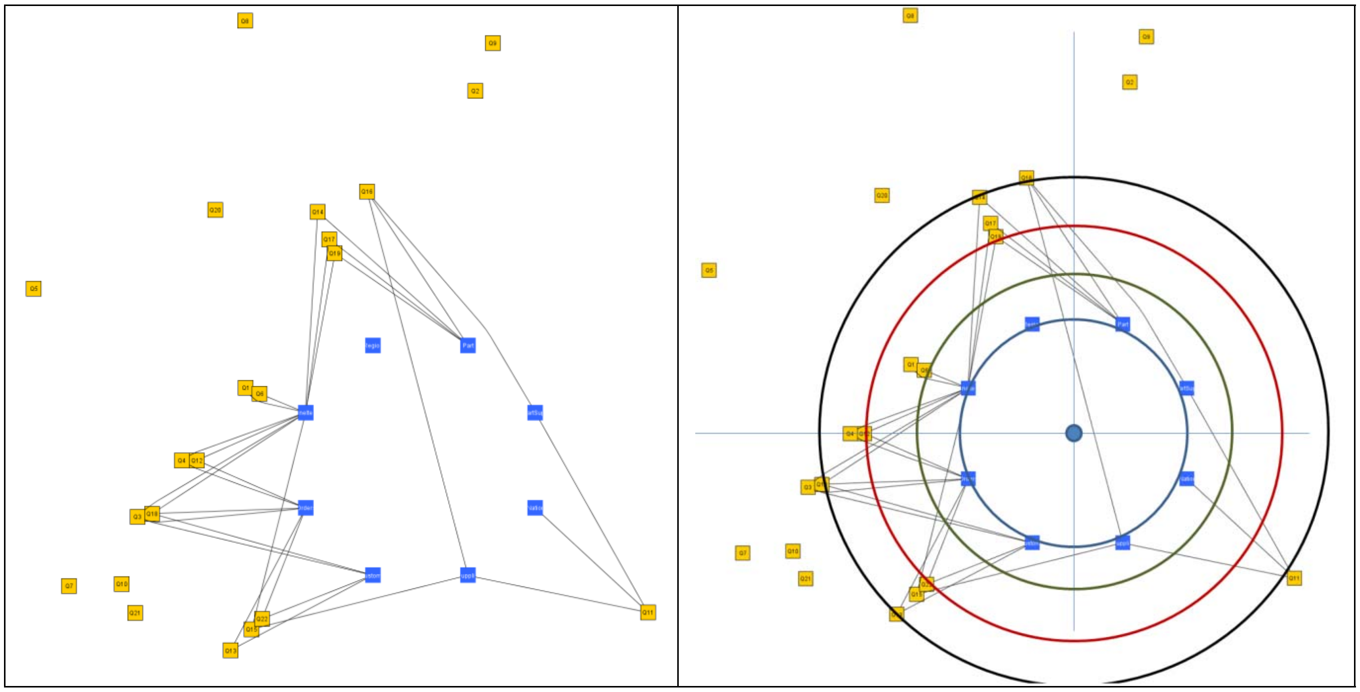
Output: an assignment of coordinates to every node v in V; specifically, for every relation r_i, a radius ρ_i and an angle θ_i, resulting in coordinates $r_i.x$ and $r_i.y$; we handle every query q_i similarly, too.

Begin

```
        $\theta_0 = 180 / |V_R|$;          // sector (pizza slice) per relation
        AlignTablesAsL1;                    // sort tables by similarity;
        For every relation $r_i$ {
                $\theta_i = \frac{\pi \cdot \theta \cdot (\frac{1}{2} + i)}{180}$
                $r_i.x = x_0 + \rho \cdot cos(\theta_i)$; $r_i.y = y_0 + \rho \cdot sin(\theta_i)$;
        }
        For every query $q_i$ {
                $\theta_{Arc}$=0
                For every relation $r_i$ s.t. $(q_i,r_i) \in E${
                        $\theta_{Arc} = \theta_{Arc} + \begin{cases} \vartheta_i, \vartheta_{Arc} = 0 \\ \vartheta_i, |\vartheta_{Arc} - \theta_i| < \pi \\ 2\pi - \theta_i, else \end{cases}$
                        $\theta_{Arc} = \theta_{Arc} / 2$ ;        //place the node in the bisector of the arc
                }
                $\theta_i = \theta_{Arc}$ ;
                $\rho_i = degree(q_i) \cdot \Delta_\rho + \rho$ ;
        }
        For every query $q_i$ {                                            //resolve conflicts
                For every query $q_k$ s.t. $conflict(q_j,q_k)$, $count_j^{conflict}$++;      //count conflicts of $q_j$
                $\rho_j = \rho_j + \cdot \Delta_\rho^{conflict} \cdot count_j^{conflict}$
                $q_i.x = x_0 + \rho_i \cdot cos(\theta_i)$; $q_i.y = y_0 + \rho_i \cdot sin(\theta_i)$;
        }
```

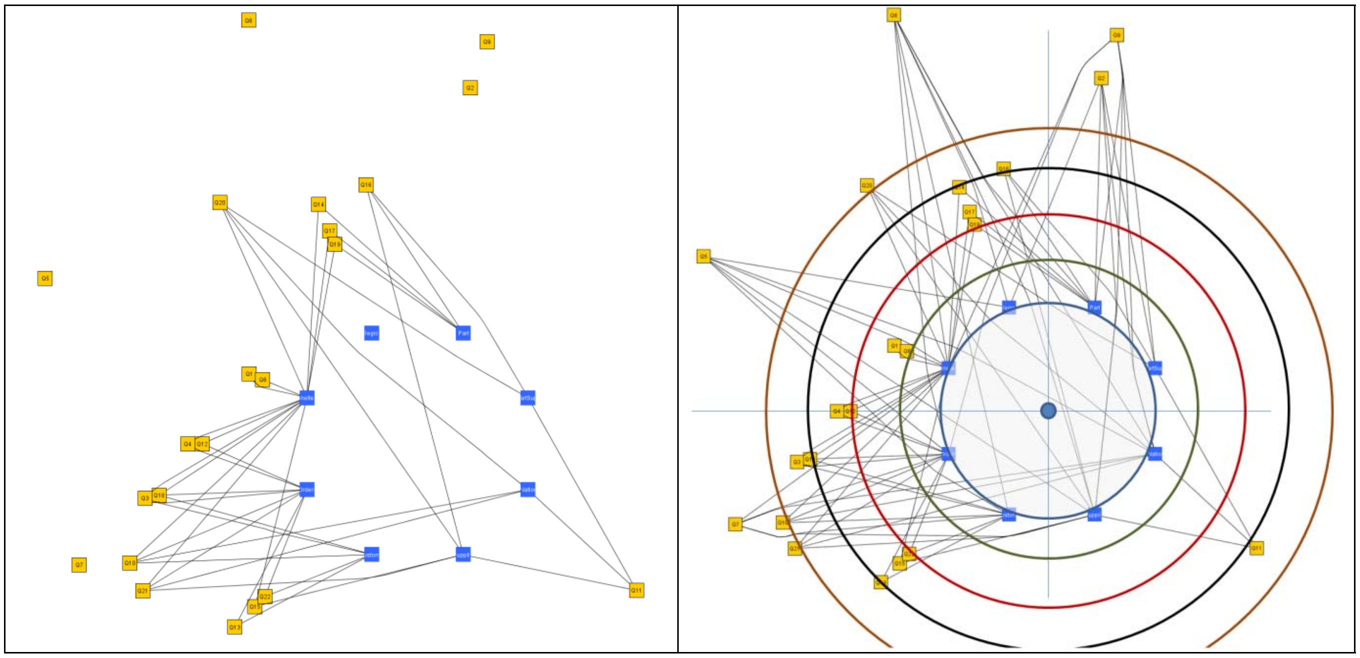
End.

Fig. 3. Algorithm *Concentric Radial.*



Maximum query fan-out: 3

Maximum query fan-out: 3, also depicting the homocentric circles



Maximum query fan-out: 4

Maximum query fan-out: 7, also depicting the homocentric circles

Fig. 4. The layout of the TPC-H ecosystem as the output of the Concentric Radial algorithm in successive steps, each with a different maximum fan-out for the query nodes. The right-side figures are also annotated with the homocentric circles to clarify the way the nodes are placed on the diagram. Edges have been smoothened with organic edge routing by the yEd tool.

In the sequel, we detail how these goals are achieved by the algorithm *Concentric Radial.*

- First the algorithm needs to align the relation nodes, as if they were about to be deployed in a line. This is done by the aforementioned *AlignTablesAsL1* procedure. The goal for this alignment is to set the related relations in close positions in the line (which will be subsequently folded to form a circle of relations).
- Then, the algorithm places every table in the appropriate slot and computes its x,y coordinates
- The third part of the algorithm deals with determining the correct angle for each query q_j. We modify the barycenter method to compute this angle. The underlying principle is to add all the angles of the relations that are accessed by a query and compute the arc that they define. However, there is a little trick here that has to do with the particularities of the circle: if two angles differ more than 180^o, then, the complement of the second must be used in order to compute a position that minimizes the arc. Once the arc of a query is computed, then, we place it in its middle (i.e., divide the overall angle by 2).
- Having computed the angles of the queries, then we set to place them in different concentric circles. For the purpose of this paper, we choose a structural criterion for the placement of the nodes and place all nodes of the same degree in the same circle, starting from nodes of fan-out 1 in the inner circle and increasing the radius with the fan-out. Again, this is a criterion that can be altered by a choice of the observer.
- The fifth step requires resolving conflicts. We define two query nodes to be in conflict if they fall too close on the diagram; formally, this means they have the same angle and their radiuses do not differ more than a predefined quantity Δ_ρ^c multiplied by the number of conflicts the node has. When this happens, the radius of the conflicting node is increased as many times as the number of conflicts it has. In other words, if a node is found to conflict with *k* other nodes, we give it an extra space of $k * \Delta_\rho^c$ radius; this will allow the rest of the *k*-1 nodes to take the respective positions (one at a time), each differing by Δ_ρ^c from the other.
- Finally, once all conflicts are resolved and the radius for all queries has been stabilized, we can easily compute the x,y coordinates for each query.

The result of the algorithm (see Fig. 6 for an anatomy of our running example) possesses several good properties: it exploits the circumference of the relations' circle, hides part of the diagram's noise (due to the overdose of edges) inside the circle and allows the progressive drawing of the diagram. The latter is most conveniently done in terms of the queries' degree.

One of the most important benefits of the algorithm, however, is that it allows the visual representation of queries that are similar to each other in an intuitive way. This is accomplished by the angle computation of the queries, along with the conflict resolution heuristics. Observe, for example queries Q14, Q17, and Q19: they all access relations PART and LINEITEM and this is clearly depicted in the diagram. The Sandwich layout is also doing a good job with this query, placing them at the rightmost end of the diagram; however, the concentric radical placement demonstrates their relationship clearly.

IV. Conclusions and Future Work

In this paper, we have provided early results towards addressing the problem of visualizing data-centric ecosystems. We have explored variants of the barycenter method (the state-of-the-art method for representing bipartite graphs) as well as the concentric radial method that appears to provide a more promising approach. This is due to two facts: First, the radial nature of the deployment allows better extensibility of the diagram and customization of the concentric circles to different application needs. Second, the method serves the purpose of spatial memory as it groups of similar queries closely in the 2D space of the canvas.

Clearly, several issues have to be further refined, including the incorporation of views in the ecosystem, the investigation of the appropriate similarity functions for queries and relations (possibly taking finer details of the queries into consideration), the combining of this logical-level representation with physical-level constructs (i.e., scripts containing the queries). Also, the anatomy of the concentric radial approach (as shown in Fig. 4) suggests there is further room of improvement in order to minimize the noise introduced by the edges and to handle significantly larger ecosystems (both in terms of queries and schemata).

References

[1] Andrew Bragdon, Steven P. Reiss, Robert C. Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, Joseph J. LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. ICSE (1) 2010: 455-464.

[2] Robert DeLine, Mary Czerwinski, Brian Meyers, Gina Venolia, Steven M. Drucker, George G. Robertson. Code Thumbnails: Using Spatial Memory to Navigate Source Code In Proceedings of the IEEE Symposium on Visual Languages and Human-centric Computing, pp. 11-18, 2006.

[3] G. Di Battista, P. Eades, R. Tamassia, I. Tollis. Graph Drawing: Algorithms for the visualization of graphs. Prentice Hall, 1999. ISBN: 0-13-301615-3

[4] Robert DeLine, Gina Venolia, Kael Rowan. Software Development with Code Maps. ACM Queue, 8(7), July 2010

[5] S.G. Eick, J. L. Steffen, E. E. Sumner Jr. Seesoft: a tool for visualizing line-oriented software statistics. IEEE Transactions on Software Engineering 18(11): pp. 957-968, 1992.

[6] Brian Johnson and Ben Shneiderman. Tree-Maps: a space-filling approach to the visualization of hierarchical information structures. In Proceedings of the 2nd conference on Visualization '91 (VIS '91), pp. 284-291. 1991. IEEE Computer Society Press, Los Alamitos, CA, USA.

[7] TPC. TPC-H benchmark, v.2.70. Transaction Processing Council, available at: http://www.tpc.org/, 2008.