Theories and Principles Matter: Towards Visually Appealing and Effective Abstraction of Property Graph Queries

JIEBING MA, Nanyang Technological University, Singapore
SOURAV S BHOWMICK*, Nanyang Technological University, Singapore
BYRON CHOI, Hong Kong Baptist University, China
LESTER TAY, Nanyang Technological University, Singapore

Existing visual abstraction of a property graph query by representing it as a *labeled atomic graph* (LAG) has great potential to democratize the usage of property graph databases as it enables user-friendly visual query formulation without demanding the need to learn a property graph query language *e.g., Cypher*. Unfortunately, existing LAG-based query interfaces do not embrace HCI principles and psychology theories to inform their design and as a result may have adverse impact on their *usability* and *aesthetics*. In this paper, we depart from the classical theory- and principles-oblivious LAG abstraction to present a novel *theory-informed* visual abstraction called *labeled composite graph* (LCG) to address this limitation. It realizes a novel and extensible *visual shape definition language* called VEDA to create and maintain an LCG systematically, guided by a variety of theories and principles from HCI, visualization and psychology. We build a novel LCG-based visual property graph query interface for *Cypher* called SIERRA and demonstrate through a user study its superiority to an industrial-strength LAG-based query interface for property graphs w.r.t. usability, aesthetics and efficient query formulation.

CCS Concepts: • **Human-centered computing** → *Visualization systems and tools*; • **Information systems** → *Query languages*.

Additional Key Words and Phrases: Visual abstraction, property graph queries, psychology, visualization, human-computer interaction, theories, principles, counterfactual thinking

ACM Reference Format:
Jiebing Ma, Sourav S Bhowmick, Byron Choi, and Lester Tay. 2023. Theories and Principles Matter: Towards Visually Appealing and Effective Abstraction of Property Graph Queries. *Proc. ACM Manag. Data* 1, 2, Article 132 (June 2023), 26 pages. https://doi.org/10.1145/3589277

1 INTRODUCTION

Abstraction is at the heart of computational thinking. Every abstraction in computer science consists of a data model and a way of manipulating the data [15]. In recent times, there is an increasing popularity of the property graph abstraction to model relationships between real-world entities. For example, Neo4j is one of the most popular property graph databases [16] that has been used in a variety of industries. A soup of query languages has been proposed to manipulate property graphs such as *Cypher* [10], PGQL [11], etc.

The increasing usage of property graph query languages in various applications demands end users to be knowledgeable of their syntax and semantics in order to formulate and interpret

*Corresponding Authors

Authors' addresses: Jiebing Ma, Nanyang Technological University, Singapore, r170022@e.ntu.edu.sg; Sourav S Bhowmick, Nanyang Technological University, Singapore, assourav@ntu.edu.sg; Byron Choi, Hong Kong Baptist University, Hong Kong, China, bchoi@comp.hkbu.edu.hk; Lester Tay, Nanyang Technological University, Singapore, lest0003@e.ntu.edu.sg.

2836-6573/2023/6-ART132
https://doi.org/10.1145/3589277

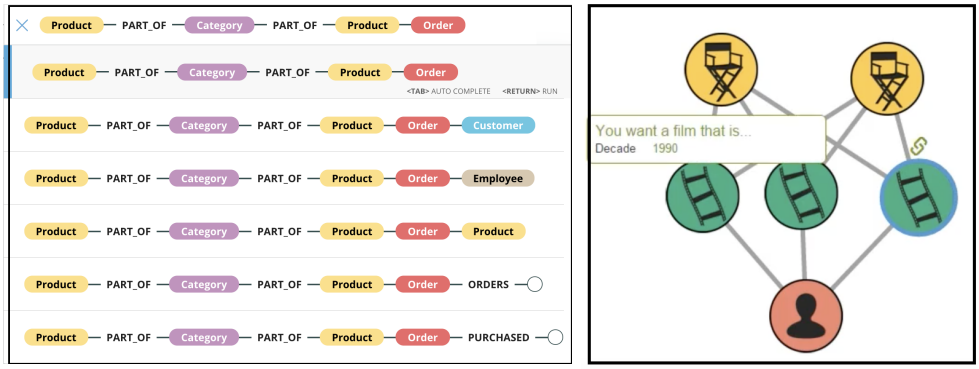

Fig. 1. LAG abstractions of Bloom [8] (left) and VISAGE [48, 49] (right).

queries correctly. In practice, such expectation makes it daunting for a non-programmer to use or understand such languages. These query languages are designed for expert users. Consequently, they lack the effective support required for easier use by domain experts who are non-programmers. Although a survey conducted by Sahu *et al.* [56] highlighted that graph query languages and usability are among the top challenges for graph processing, majority of research in academia and industry have primarily focused on addressing expressiveness, efficiency and scalability issues associated with property graphs and its query languages. In this paper, we depart from this tradition by focusing on the usability challenge.

The LAG abstraction. A popular starting point for mitigating the usability challenge is to create a visual abstraction of a query by representing it as a *labeled atomic graph* (LAG). In this abstraction, the query nodes are represented using *atomic* shapes (*e.g.,* circle) and the links are shown as directed or undirected *atomic* (labeled) edges. It may also support operations that color code the nodes or map their labels to icons. The LAG abstraction can be then leveraged on a visual query interface (a.k.a GUI) to facilitate a user to draw a property graph query interactively by utilizing *direct-manipulation* [57]. It can also enable one to comprehend a textual query easily by mapping it to its visual counterpart.

Example 1. Several commercial and academic property graph GUIs such as Neo4j's Bloom [8], VISAGE [48, 49] have embodied the LAG abstraction. Specifically, Bloom allows users to search with *type-ahead suggestions* where one is prompted to either input a node label or a relationship type, with a list of suggestions displayed as a drop-down menu (Figure 1 (left)). Upon choosing an option from this menu, the suggestion list is dynamically updated based on current input. The LAG abstraction of Bloom is a color-coded graph with labels on nodes and edges. VISAGE, on the other hand, supports direct manipulation-based query formulation and an example of its LAG abstraction is depicted in Figure 1 (right). Observe that it is also a color-coded graph where node labels are mapped to icons. ∎

Limitations of the LAG abstraction. Despite its simplicity and popularity, a major limitation of the LAG abstraction is that its representation of predicates on query nodes or edges can be unpalatable to downstream applications centered around property graphs. The LAG strategy of explicitly labelling these nodes or edges with multiple predicates or making the predicates available only on-demand (*e.g.,* when a node or edge is clicked or hovered on) makes the visual abstraction misaligned with theories and principles in HCI, visualization and psychology (*e.g.,* Gestalt's principles [54], *Berlyne's aesthetic theory* [20]) that are essential for designing user-friendly

and aesthetic GUIs. Specifically, it adversely impacts *visual* and *task complexities* of a LAG-based GUI. The former can be defined as "the amount of variety in a stimulus pattern" [19] whereas the latter can be defined as *resources* (*e.g.,* cognitive demands, physical and mental demands, short term memory requirements) requirement of human information processing [35].

In particular, visual complexity is typically postulated to be the integration of multiple dimensions or facets such as *quantity of information*, *variety of visual form* (*e.g.,* color, size), *spatial organization* (*e.g.,* symmetry), and *perceivability of details* (*e.g.,* congestion, amount of white space) [41]. Research in psychology have discovered a strong inverted U-shaped relationship between aesthetics and visual complexity [19, 20]. It is worth noting that any visual interface/abstraction must pay attention to its visual appearance (*i.e.,* aesthetics) as it is an important factor that impacts its usability (*i.e., aesthetic-usability effect* [1]).

Unfortunately, the textual labelling of nodes or links with predicates increase cluttering around them, adversely impacting the visual complexity and aesthetics [40, 41] of any LAG-based GUI. On the other hand, concealing the predicates by default prevent a user from getting a bird's-eye view of predicates imposed on different query nodes and edges without explicitly clicking or hovering on them. Additionally, the LAG abstraction does not provide effective visual cues to properties that are involved in predicates. This increases task complexity in any LAG-based GUI as it is inefficient to identify relevant predicates at a glance in a visual representation of a property graph query. Furthermore, it may increase the short-term memory load on a user for various downstream tasks (*e.g.,* visual query formulation, query interpretation) as one has to retain in memory the properties that have predicates. That is, the classical atomic representation of nodes and edges in a LAG abstraction limits effective visualization of predicates on their properties.

Lastly, the operations on nodes and edges in a LAG are *data-unaware* and "static" as they do not "optimize" their shapes and sizes to visually convey information associated with them effectively. Typically, all query nodes or links have the same size and shape regardless of the number of predicates imposed on them. For instance, one node may have four predicates and another may have one. However, it is not possible to know this difference in number by simply eyeballing the LAG representation of the query.

Example 2. Reconsider the LAG abstraction of VISAGE in Figure 1 (right). We note the clip icon beside the green node, which conveys that one or more predicates are associated with it. A user can view them by clicking on the clip. Also, all nodes have the same size and shape. Hence, it is neither possible to get a bird's-eye view of the properties (*i.e.,* attributes) involved in the predicates on the green node nor the number of predicates associated with it by simply eyeballing the LAG view. Similarly, the visual representation of a property graph query in Bloom does not reveal any predicate-related information at a glance. ∎

Do these limitations of LAG matter to end users? To answer this question, we surveyed 17 end users of Bloom [8] in the context of visual query formulation. Our survey focused on whether a set of features that are adversely impacted by the LAG abstraction is indeed important to them. As detailed in Section 5.1, our engagements with them reveal that majority think support of these features are paramount for visual formulation of *Cypher* queries. It also highlights the importance of usability and aesthetics.

Our contributions. In this paper, we present a novel visual abstraction of property graph queries called *labeled composite graph* (LCG) to address the aforementioned challenges. Specifically, the data model of LCG exploits *derived shapes* to represent query nodes and links in order to visually represent node and edge predicates effectively. Intuitively, a query node is represented using a color-coded circle with a set of smaller color-coded circles (possibly empty) on its circumference to represent predicates on properties. The color of a circle has one-to-one mapping with the color of

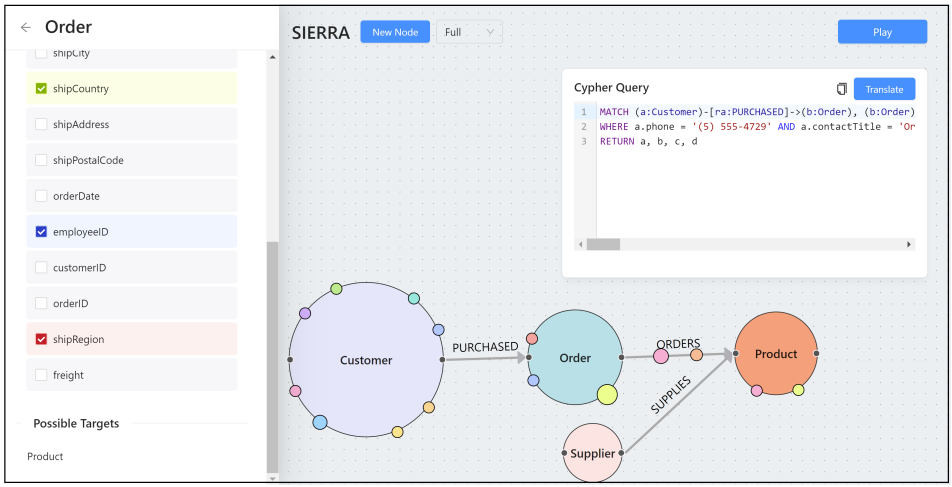

Fig. 2. The LCG abstraction of SIERRA.

the corresponding node label or property. Similarly, a set of color-coded circles is overlaid on a directed edge to represent predicates on links. An example of the LCG data model is depicted in Figure 2. Consider the ORDERS link. By simply looking at the color codes of the circles, we know that it has predicates on two properties (*i.e.,* two circles on the edge), namely orderID and unitPrice. Furthermore, observe that the Customer node is larger than the Order node, indicating that a larger number of predicates is imposed on the former compared to the latter. Hence, in contrast to the LAG abstractions in Examples 1 and 2, the composite representation and visual cues in an LCG enable a user to gain a richer view of the predicates on various nodes and links without any additional interaction.

At first glance, it may seem that the creation of the LCG abstraction is straightforward. However, heedless creation of circles with arbitrary radius and positioning them randomly on a circumference or an edge may compromise visual complexity and aesthetics (*e.g.,* two circles may overlap) of any LCG-based GUI due to its impact on symmetry and congestion [40, 41]. Hence, these circles need to be *positioned* judiciously and their sizes need to be *maintained* in a *data-driven* manner. To this end, we present a novel and extensible *Visual shape Definition Language (VEDA)* that implements a set of *visual shape operators* to manipulate various shapes in an LCG in a principled manner. Specifically, these operators exploit principles and theories from HCI, visualization and psychology to *automatically generate*, *maintain* and *position* these shapes to balance visual and task complexities, usability, and aesthetics. As a proof of the effectiveness of the LCG abstraction, we implement a novel LCG-based visual property graph query interface for read-only *Cypher* queries coined SIERRA (viSual Interface for quErying pRoperty gRAphs).

A cornerstone of our proposed abstraction is that its design is *informed* by theories and principles in HCI, visualization and psychology (detailed in Section 3.2). Despite the importance of design guided by theories and principles being foundational to designing technology for human use, theory-informed design in HCI is surprisingly rare [45]. In this paper, we take a concrete step towards theories- and principles-driven design and demonstrates its usefulness in the context of property graph queries.

Our user study detailed in Section 6 demonstrates the superiority of the LCG abstraction over the LAG abstraction in the context of visual query formulation. Specifically, it validates the importance of the novel aesthetics and usability-conscious features of the LCG design and shows the positive

impact they have on the participants of the study. Furthermore, our LCG-based visual query interface (*i.e.,* SIERRA) facilitates faster query formulation compared to a popular LAG-based commercial visual interface for property graphs (*i.e.,* Bloom).

In summary, this paper makes the following key contributions.

- Guided by theories and principles of psychology, HCI and visualization, we propose a novel visual abstraction of property graph queries called *labeled composite graph* (LCG).
- We present a novel data-driven framework grounded on the *visual shape definition language* (VEDA) for generating, maintaining and positioning the components of an LCG in a principled manner to preserve a balance between visual and task complexities and aesthetics.
- We describe a novel LCG-based GUI called SIERRA for formulating read-only *Cypher* queries effortlessly.
- We present a user study to demonstrate the superiority of the LCG abstraction to its LAG counterpart w.r.t. usability and aesthetics.

The rest of the paper is organized as follows. We present background information necessary for understanding the LCG abstraction in Section 2. Section 3 introduces the LCG abstraction. We present the VEDA framework to realize the LCG abstraction in Section 4. We describe the SIERRA GUI based on the LCG abstraction in Section 5. Section 6 reports the performance and effectiveness of the proposed abstraction. We review related work in Section 7. The last section concludes the paper.

2 BACKGROUND

In this section, we first give a brief introduction to the property graph model and the read-only *Cypher* query language using examples. In this paper, we use *Cypher* as the representative query language of property graphs as its implementation is not limited to the Neo4j database but also has been incorporated in other software such as SAP HANA Graph [46], RedisGraph [13], and Memgraph [7]. It has also been adopted in Cypher for Apache Spark [2] and Cypher over Gremlin [3]. Next, we briefly introducing relevant HCI, visualization and psychology principles and theories that guide the design of the LCG abstraction.

2.1 Property Graph Model and Cypher

A property graph data model (PGDM) aims to organize and manage graph data comprising of *nodes* and *relationships*. Nodes typically represent real-world entities whereas relationships model connection between them. A distinguishing feature of PGDM is that the nodes and relationships also store additional information in the form of *node labels*, *relationship types*, and (node or relationship) *properties*. Specifically, each node in a property graph may have any number of *labels*, denoted by a leading colon. Note that labels are optional. Each relationship is directed and involves a source node, a target node, and a type. The type is denoted by a leading colon. Note that it is possible in PGDM to have multiple relationships having same types and properties between same pair of nodes. Figure 3 depicts an example of a property graph involving suppliers, customers, and products. Observe that the node and relationship labels have a leading colon (*e.g.,* :Product, :SUPPLIES). Nodes and relationships may have an optional *property map* in the form of a set of key-value pairs (*e.g.,* :Order node has orderID and orderDate properties).

We now briefly describe the basic features of the *read-only Cypher* query language. A detailed exposure of *Cypher* can be found in [28]. We use the MATCH-WHERE-RETURN clause of read-only *Cypher* as a representative property graph query language. It exploits an "ASCII art" (*i.e.,* images created using text characters) representation of nodes and relationships. For example, the following query can be used to retrieve all customers from US cities excluding San Francisco and Seattle who

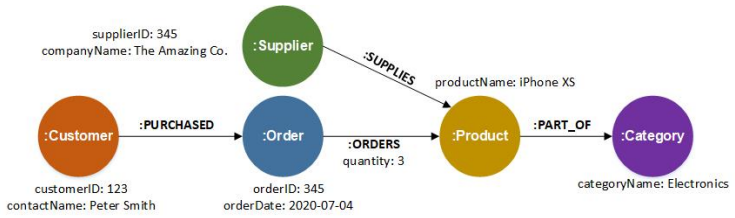

Fig. 3. An example of property graph model.

purchased two orders involving cheese products. One of the order involves more than 20 quantities at a 10% discount whereas the other one has no discount.

```
 MATCH (a:Customer)-[ra:PURCHASED]->(b:Order),
(a:Customer)-[rb:PURCHASED]->(c:Order),
(c:Order)-[rc:ORDERS]->(d:Product),
(b:Order)-[rd:ORDERS]->(e:Product),
(e:Product)-[re:PART_OF]->(f:Category),
(d:Product)-[rf:PART_OF]->(f:Category)
 WHERE a.country = 'USA' AND a.city <> 'San Francisco' AND
  a.city <> 'Seattle' AND f.description = 'Cheeses' AND
  rc.quantity > 20 AND  rc.discount = '0.1' AND rd.quantity > 20
RETURN a
```

Cypher queries consist of a sequence of clauses. These clauses are evaluated on a property graph. The MATCH clause allows one to specify the patterns to search for in the underlying property graph. Note that a node in a pattern may not have a label (*e.g.,* d instead of d:Product) and a relationship can be of variable length with properties defined on it. The WHERE clause acts as a filter (*i.e.,* predicates) to the MATCH patterns to make them more specific. In the above example, this would remove all matches where the customer is from San Francisco or Seattle. Finally, the RETURN clause projects all records to the variable(s) given in the clause. For example, we would end up with a result containing records associated with the variable *a*.

2.2 Principles and Theories

In this work, we assume that the property graph queries are generated by humans. Any visual abstraction of such queries should enable them to understand what they see–and find what they want–at a glance. It should be aesthetically appealing. Consequently, its design should be informed by theories and principles from HCI, visualization and psychology. There are numerous theories and principles in these areas. Which ones should we leverage? Broadly, our selection should be guided by the knowledge and insights they provide on visual and task complexities, usability and aesthetics as well as their adoption in related research or in practice. In this section, we briefly describe and justify the non-exhaustive set of relevant theories and principles we exploit in this work.

Gestalt principles. They describe how humans group similar elements, recognize patterns and simplify complex images [5, 54]. They are essential for making visual abstractions aesthetically pleasing and easy to understand. In fact, Gestalt principles are widely adopted in aesthetic web page design [24] as well as in SQL query visualization [38]. The principles that are relevant to the LCG abstraction are the laws of *proximity*, *similarity*, *closure*, *symmetry*, and *continuity*. The law of *proximity* states that "when an individual perceives an assortment of objects, they perceive objects that are close to each other as forming a group" [5]. The law of *similarity* states that

Table 1. **HCI, psychology and visualization principles and theories in the design of the LCG abstraction and SIERRA GUI.**

Principles & Theories	Representative features of LCG	Representative features of SIERRA GUI
Gestalt Principles *[5, 54]*		
Law of proximity	Label group, property groups, predicate circle set along circumference.	Properties of a node/link in the left panel are grouped together.
Law of similarity	Predicate circles are all circular shape, nodes are circular shape, edges are directed arrows.	
Law of closure	Circular shape of query nodes and predicates.	
Law of symmetry	Predicate circles are symmetrically positioned along a circumference or directed arrow.	
Law of continuity	Predicate circles are aligned on the circumference or arrow.	Node labels, properties are aligned and grouped together in the left panel.
Visual Complexity		
Berlyne's aesthetic theory [19, 20]	The LCG abstraction adds a moderate degree of visual complexity that has positive impact on aesthetics (Section 6).	
Quantity of information [25, 27, 41, 42, 44, 60]	The numbers of predicate circles and color codes are kept to minimum to sufficiently convey relevant information visually.	Details of predicates/node labels/relationships are exposed on-demand.
Variety of visual form [25, 41, 44, 50, 55]	All nodes and predicates have a consistent circular shape. Colors are used to facilitate visual correlation. Size dissimilarity is only used to inform relative number of predicates on properties and node labels.	Consistent text format to list properties in the left panel. Only properties selected for predicates are color coded, mitigating the impact of color variety.
Spatial organization [41, 42, 44]	Positioning and alignment of predicate circles along circumference/arrow to maintain symmetry.	Position and alignment of properties of a node label/relationship in the left panel.
Perceivability of details [39, 41]	No visual congestion due to maintenance of distance between predicate circles and concealment of predicate details, colors in predicate circles and nodes have clear contrast with the background.	Colored text of properties in the left panel is correlated with corresponding colored predicate circles.
Shneiderman's Golden Rules *[57]*		
Strive for consistency	Consistent representation of nodes, links, and predicates.	Consistent sequence of actions to create any query.
Offer informative feedback		For every visual action, there is a system feedback on its completion.
Offer simple error handling		Easy to handle errors through easy reversal of actions. SIERRA also provides list of nodes and relationships, property map, and possible target nodes (left panel) so that users can choose them instead of manually entering the data. Manually entered values are also checked for valid labels. Predicate values can also be chosen from drop-down list to minimize errors.
Permit easy reversal of actions		Deletion or update of nodes, edges, or predicates can be easily done with at most two actions.
Support internal locus of control		Users of SIERRA are the initiators of actions during query formulation.
Reduce short-term memory load	Details on demand and color-coding of labels, properties, and circles reduce short-term memory load.	Color-based map of properties between left and middle panel eliminates the need to remember color-property map.
Visualization		
Expressiveness criteria [37]	The circles, arrows, and colors capture exactly the facts, no more, no less.	
Effectiveness criteria [23, 37]	Uses color, position, and shapes which are high/medium-ranked in perceptual tasks list of nominal data type.	
Principle of Importance Ordering [37]	Area is used to convey relative differences in the number of predicates. This is the most effective choice compared to other higher-ranked alternatives in the perceptual tasks list.	

"elements within an assortment of objects are perceptually grouped together if they are similar to each other" [5] (*e.g.,* in the form of shape, colour, shading etc.). The law of *closure* embodies that "humans tend to perceive objects as complete rather than focusing on the gaps that the object might

contain" [5]. For example, a circle has good Gestalt based on the law of closure [5]. The law of *symmetry* states that "the mind perceives objects as being symmetrical and forming around a center point" [5]. Therefore, the mind perceptually connects two disconnected symmetrical elements in order to form a coherent shape. Lastly, according to the law of *continuity*, "elements of objects tend to be grouped together, and therefore integrated into perceptual wholes if they are aligned within an object" [5].

Eight golden rules of interface design. Shneiderman proposed a collection of heuristic principles that is widely applicable in most interactive systems [57]. These rules are strive for consistency, enable frequent users to use shortcuts, offer informative feedback, design dialogue to yield closure, offer simple error handling, permit easy reversal of actions, support internal locus of control, and reduce short-term memory load. These principles are widely used for usable interface design [61].

Principles and theories of visual complexity and aesthetics. Visual complexity increases cognitive load [31] and decreases visual aesthetics and usability [39, 59]. Since the visual abstraction of a property graph query is typically displayed on a visual interface, we leverage recent research in HCI and GUI design that define and characterize visual complexity of a GUI [41]. It is operationally defined as the combination of different features such as *quantity of information*, *variety of visual form*, *spatial organization*, and *perceivability of details* [41]. *Quantity of information* is considered to be the most common facet of visual complexity: the more units of information are on the screen, the more complex it appears to a user (in psychology, it is referred to as *set-size effect* [60]). In the literature, it is also referred to using several synonymous terms such as amount of elements [42], quantity of objects [44], detail of information [25], and amount of detail or intricacy [27]. *Variety of visual form* involves facets such as visual diversity [25], dissimilarity/irregularity of objects [50], and color variety [44]. Consequently, it embodies the number of colors, shapes, sizes, background textures and other visual features used to represent information. It has been observed that when these numbers increase the complexity increases as well [55]. *Spatial organization* refers to "the tendency of human perception to see structural repetition and regular positioning as simplifying presented information" [41]. In the literature, it is also referred to as organization [44] and disorganization [42]. *Perceivability of details* [39] reflects "the limitations of human visual perception, such as needing to use the focal vision to perceive fine-grain detail or struggling to efficiently distinguish low-contrast items from background" [41]. Importantly, a single feature (*e.g.,* quantity of information) is not sufficient or practical in understanding visual complexity [41]. Otherwise, an empty LAG will be considered as a best design w.r.t. visual complexity.

Visual complexity influences aesthetics. The relationship between them is well established in psychology [19, 20, 53] as well as in HCI [59]. Several of the aforementioned features of visual complexity are exploited to quantify aesthetics of visual interfaces [40, 51]. In particular, *Berlyne's aesthetic theory* [19, 20] is an influential theory in psychology which states that the relationship between visual complexity and aesthetics follows an inverted U-shaped curve where stimuli of a moderate degree of visual complexity is considered palatable but both less and more complex stimuli are considered unpleasant. Note that aesthetics is a complex phenomenon [18] consisting of many culture-independent and culture-specific dimensions. Berlyne's theory aims to only focus on culture-independent aspects related to visual complexity [52]. *Hence, any visual abstraction should balance visual complexity with aesthetics.*

Principles and theories of visualization. Since visual abstractions are graphical presentations of data, theories and principles related to *expressiveness* and *effectiveness* criteria for graphical languages [37] are relevant to our work. These criteria are also exploited recently in SQL query visualization [29, 38]. The former criteria determines whether a graphical language can express

the desired information whereas the latter "identify which of these graphical languages, in a given situation, is the most effective at exploiting the capabilities of the output medium and the human vision system" [37]. Specifically, Cleveland and McGill [23] observed that people execute the perceptual tasks associated with the interpretation of graphical presentations with different degree of accuracy. Based on this observation, Mackinlay [37] identified and ranked perceptual tasks for encoding quantitative, ordinal, and nominal data. In particular, position is ranked the highest for all types of data whereas color is ranked high for both ordinal and nominal data (*i.e.,* effective way of encoding them). Area is moderately effective for encoding quantitative data. Based on this ranking, Mackinlay [37] proposed the *Principle of Importance Ordering*: Encode more important information more accurately (*i.e.,* use tasks higher in the ranking to encode more important information).

3 THE LCG ABSTRACTION

We begin by presenting briefly the design philosophy behind the LCG abstraction. Next, we introduce the data model of LCG and present how the aforementioned theories and principles of HCI, visualization and psychology manifest in it. Finally, we describe a language called VEDA to create and manipulate an LCG in a principled manner grounded on these theories and principles.

3.1 Design Philosophy

The issue of how theories and principles inform design is fundamental to designing technology for human use [45]. However, recently Oulasvirta and Hornbaek [45] lamented that theory-informed design in HCI is surprisingly rare. According to them, for a theory to be considered a "theory for design", it must advance design and attain *good* features. Specifically, theories and principles can direct design choices, address design problems by identifying the best choice among a finite collection of options, and facilitate rethinking of design problems by revealing new design spaces.

Inspired by the notion of "theory for design", our design philosophy is grounded on the aforementioned theories and principles. Concretely, it aims to achieve the followings.

- **Effective and aesthetic visual representation.** The LCG abstraction should support *effective* and aesthetically pleasing visual representation of predicates as well as the query. A user should should be able to see and find, at a glance, information associated with query nodes and links.
- **Effective support of downstream applications.** The LCG abstraction should facilitate user-friendly formulation of any read-only property graph query in a GUI. It is also desirable for an LCG-based GUI to support efficient formulation as measured using query formulation time.

3.2 The Data Model

A *labeled composite graph (LCG)* is used to visually represent a property graph query. It comprises of two *types* of visual shapes, *elementary* and *derived*, for representing nodes, links and predicates in a query. The nodes and directed links in an LCG correspond to the nodes and relationships in the clauses of the *Cypher* query, respectively (Section 2.1). For clarity, we ignore the leading colons in node labels and relationship types. The predicates correspond to the predicates in the WHERE clause. The nodes involved in the RETURN clause are shown with bold outline. Specifically, the nodes in an LCG are color-coded and represented by *circle* and *composite circle*. Similarly, the edges are represented by an *arrow* and a *composite arrow*. The circle (resp. arrow) represents a node (resp. relationship) in a query graph without any predicates on its properties. The composite circle (resp. arrow) models a node (resp. relationship) with at least one predicate on its properties where a set of smaller color-coded circles is overlaid *systematically* on its circumference (resp. length) to represent

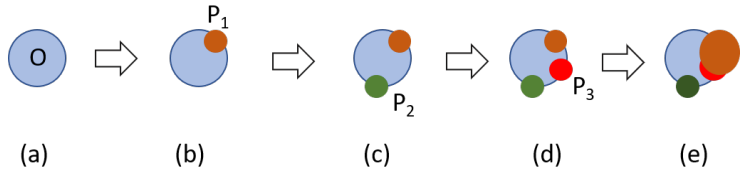

Fig. 4. Evolution of a composite circle. (a) No predicate; (b)-(d) Three predicates on three properties; (e) Two additional predicates on the property represented by the brown predicate circle.

them. Note that the circle and arrow belong to the elementary shape type whereas the composite circle and arrow are derived shapes. The color of a circle has one-to-one mapping with the color of the corresponding node label or property. Given a composite circle (resp. arrow), we refer to the circle (resp. arrow) representing a node (resp. relationship) and one associated with a property as *parent circle (resp. arrow)* and *predicate circle*, respectively. Observe that when a query does not have any predicates, the LCG is indistinguishable from its classical LAG counterpart containing only circles and arrows.

Figure 2 depicts an example of LCG. Consider the Order node. By simply eyeballing it and looking at the color codes of the predicate circles, we know that it has predicates on three properties (*i.e.,* three predicate circles), namely shipCountry, shipRegion and employeeID. The shipCountry predicate circle (yellow node) is larger than the rest since it has multiple predicates defined on it. These predicate circles are positioned systematically (discussed later) to ensure that the LCG is aesthetically pleasing. Furthermore, the size of a query node is proportional to the number of predicates associated with its properties. For instance, the Customer node is the largest since it has the largest number of predicates. On the other hand, Supplier is the smallest node since it has no predicate imposed on its properties. Observe that in contrast to the LAG abstraction, the visual cues in an LCG enable a user to get a bird's-eye view of the predicates on various nodes and links.

We now elaborate on how these features embrace the aforementioned theories and principles (first two columns of Table 1). Consider the Gestalt principles. The grouping of predicate circles on a circumference or arrow is aligned with the principles of proximity and continuity. The circular shape of predicate circles and their systematic positioning on the circumference and length are guided by the principles of similarity, closure and symmetry.

The LCG is designed to moderately increase the visual complexity compared to a LAG but enhance aesthetics (demonstrated in Section 6). This is consistent with Berlyne's aesthetic theory [19, 20]. Specifically, the existence of predicate circles, colors and different sizes increase visual complexity. The number of predicate circles in a parent circle/arrow and the number of distinct colors depend on the number of properties involved in predicates. Hence, the quantity of information and variety of visual form facets are proportional to them. Observe that we use the predicate circle size to convey multiple predicates on a given property instead of creating same-size circles for each predicate. Furthermore, we use unique color codes for predicate circles, eliminating the need to label them textually which increases visual clutter. Similarly, the number of distinct colors of nodes in a query depends on the number of distinct node labels and not on the number of nodes (*e.g.,* four distinct node labels in Figure 2 have four colors). Also, the size dissimilarity of circles is only used to provide visual cues to different numbers of predicates associated with node labels or properties *relatively* (*i.e.,* the exact number is not important). Given that queries are formulated by end users, the numbers of distinct node labels and predicates in an LCG are modest. Hence, the LCG abstraction moderates the impact of quantify of information and variety of visual form on visual complexity. The color choices, automatic maintenance of distance between predicate circles and elimination of

their text labels are features influenced by the perceivability of details facet. Lastly, as we shall see later, the predicate circles are systematically laid out to optimize their spatial organization.

Most of the Shneiderman's golden rules are primarily used for GUI design (detailed in Section 5.2). The LCG exploits the strive for consistency and short-term memory load reduction rules. The former is ensured by consistent representation of nodes, links, and predicates for any query. The latter is realized by explicitly representing predicates on different properties using unique color-coded circles. In this way, a user does not need to remember which properties are involved in predicates. This is superior to the clip icon used by VISAGE to represent *all* predicates on a node.

Lastly, the LCG abstraction is grounded on the expressiveness and effectiveness criteria of visualization [37]. Each parent circle (resp. arrow) exactly describes the corresponding query node (resp. relationship). Each predicate circle exactly describes predicate(s) on a specific property of a node or relationship. Color codes are uniquely paired with corresponding property names. Hence, these components do not encode additional, possibly incorrect facts. Removal of any of them would lead to incomplete visualization of the query features. The effectiveness criteria are used to determine the most effective design. Since the nodes, links and predicates are nominal data type, position, color and shape are chosen as they have high or moderate ranking in the list of perceptual tasks. The principle of importance ordering is addressed by ensuring that "best" possible perceptual task is used to encode more "important" information. In particular, area is used to quantify the number of predicates associated with circles (*i.e.,* quantitative data). This is because perceptual tasks higher in the ranking (*e.g.,* length, angle, slope) are not suitable to effectively represent this information.

3.3 Visual Shape Definition Language (VEDA)

We now introduce a novel and extensible *visual shape definition language*, called VEDA[1], for automatically creating and maintaining the components of an LCG in a principled way that embrace our design philosophy. We begin by motivating the need for VEDA.

Motivation for VEDA. As remarked earlier, classical LAG representation is not effective for property graph queries with multiple predicates on node and relationship properties. The LCG abstraction mitigates this problem by incorporating derived shapes represented using composite circles and arrows. However, it introduces challenges w.r.t. balancing visual complexity and aesthetics. We elaborate on it with an example in the context of query formulation.

A node may have multiple predicates on one or more properties. These predicates are added or modified iteratively on a GUI during query formulation. Let us assume that the parent circle's radius is fixed (as in a LAG abstraction) and the predicate circles are positioned *randomly* on the circumference. Figure 4 depicts the evolution of a composite circle with the addition of predicate circles. Specifically, Figures 4(b)-(e) depict possible visual representations of a composite circle. It is easy to see that such random positioning may lead to visual congestion, lack of spatial symmetry, as well as insufficient space to add new predicate circles (*e.g.,* Figure 4(e)). Similar situation may arise for a composite arrow as well. Hence, this may violate aforementioned Gestalt's, visual complexity and visualization principles, adversely impacting the usability and aesthetics of the visual abstraction. At this point, one may argue that a user may *manually* move around the predicate circles along the circumference to address this issue. However, such manual engagement increases task complexity by enforcing users to partake in unnecessary interactions, thereby increasing query formulation time. Hence, it is paramount to *automatically maintain* these derived shapes by embracing aforementioned principles and theories. We achieve this through VEDA. Note that

[1] *Veda* in Sanskrit means "knowledge". We aspire to create the language guided by the knowledge and insights in aforementioned theories and principles.

Table 2. **Alphabet of VEDA.**

Symbol	Description	Visual Actions for Query Formulation
Circle	A circle to represent a node or a predicate in an LCG.	add(q,c) and update(q,c) where c is a node or predicate.
Arrow	An arrow to represent a relationship.	add(q,c) and update(q,c) where c is a relationship.
Appears	Existence of a component in an LCG for the first time.	add(q,c)
Disappears	Transition from an existence of a query component in an LCG to its disappearance.	delete(q,c)
Bold	Outline a circle from unbold to bold.	return(q,c)
Empty	A query component is empty.	delete(q,c)

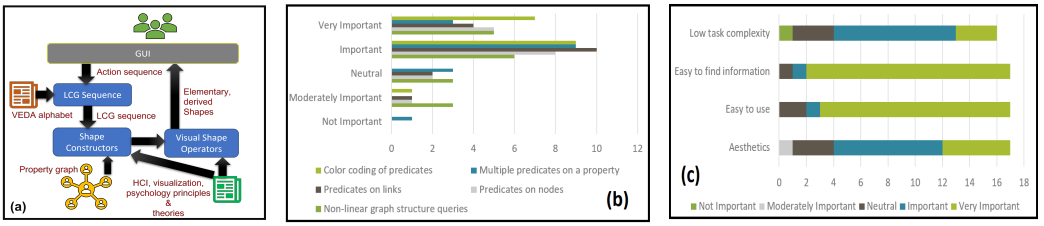

Fig. 5. (a) Overview of VEDA; (b) - (c) User survey on desirable features in *Cypher* GUI.

graphical languages for relational data such as APT [37] or state-of-the-art visual abstractions of SQL queries [29, 38] do not address this challenge posed by these derived shapes w.r.t. visual complexity and aesthetics.

Overview of VEDA. Strings are fundamental building blocks in computer science and are defined over any non-empty finite set called *alphabets*. The members of the alphabet are the *symbols* of the alphabet. A *string over an alphabet* is a finite sequence of symbols from that alphabet. A *language* is a set of strings.

In a similar vein, a *VEDA alphabet* (alphabet for brevity) is a finite set of *symbols*. Intuitively, the symbols here are the elementary shapes (*i.e.,* circle, arrow) in an LCG and possible *states* of these shapes. Note that during creation and maintenance of an LCG, the *states* of a shape can take any of the three forms: appears (*i.e.,* comes into existence), disappears (*i.e.,* disappears due to deletion), and bold (*i.e.,* a shape changes to bold outline). There is also a special state, empty, that represents an empty LCG. An *LCG sequence* over the alphabet is a finite sequence of symbols from the VEDA alphabet.

Observe that the alphabet excludes derived shapes whose construction is query-dependent and should be judiciously undertaken to embrace the aforementioned theories and principles. It also does not assign colors to them which is data-dependent. Hence, although an LCG sequence can be a basis for constructing a LAG representation of a query, it is insufficient to construct and maintain an LCG. To address this, we propose *shape descriptors* over the alphabet and *visual shape operators*. The former allows us to *define* color-coded elementary and derived shapes in an LCG and their positions on a GUI. An LCG sequence of a query is automatically transformed to a sequence of shape descriptors for a given GUI and data. The latter constructs, maintains and renders the shapes in a systematic way based on the shape descriptors. Specifically, they create the graphical design of an LCG (*i.e.,* visual abstraction) and an image rendered from that design. Our primary concern in this paper is the graphical design. We now elaborate on these concepts. Note that VEDA (Figure 5(a)) is *extensible* as it can easily be augmented to support different symbols and shapes.

VEDA alphabet. The syntax for specifying VEDA alphabet is: (alphabet (symbol n)). Here *symbol* is a symbol of the alphabet being defined and n is the *name* of the symbol representing elementary

Table 3. **Shape descriptors using VEDA alphabet.**

Descriptor Name	Definition	Shape Type	Purpose
Circle	(shape circle($color$, r, x, y, s) appears circle n)	Elementary	Circle shape to represent a node or predicate.
Arrow	(shape arrow($color$, $width$, x_1, y_1, x_2, y_2) appears arrow n)	Elementary	Arrow shape to represent a relationship.
Bold	(shape bold($color$) bold n)	Elementary	For specification of return clause.
Empty	(shape empty() empty| disappears circle n | disappears arrow n)	Elementary	Initialize or removes a shape.
Comp_circle	(shape comp_circle(D_n) join appears circle n appear circle p_n ...)	Derived	For generating composite circle representing a node and predicate(s).
Comp_arrow	(shape comp_arrow(D_n) join appears arrow n appear circle p_n ...)	Derived	For generating composite arrow representing a relationship and predicate(s) on it.
Expand	(shape expand() up circle p_n)	Derived	Increase the size (resp. width) of a composite circle (resp. composite arrow) with addition of new predicate.
Shrink	(shape shrink() down disappears circle p_n)	Derived	Decrease the size (resp. width) of a composite circle (resp. composite arrow) with deletion of an existing predicate.

n=node or relationship, p_n=predicate of n

shapes and corresponds to a node label, relationship, predicate property name, or '*' (*i.e.,* arbitrary node/relationship). Table 2 (first two columns) describes the alphabet.

LCG sequence. Given the alphabet, we illustrate the notion of LCG sequence with an example in the context of query formulation. Consider the formulation of the Order-Product subgraph in Figure 2. An LCG sequence $\mathbb{L}$ to represent it is as follows: (empty appears circle *Order* appears circle *Order.shipCountry* appears circle *Order.shipRegion* appears circle *Order.employeeID* circle *Order.shipCountry* circle *Order.shipCountry* appears circle *Product* appears arrow *ORDERS* appears circle *ORDERS.orderID* appears circle *ORDERS.unitPrice* circle *ORDERS.orderID* appears circle *Product.supplierID* appears circle *Product.unitPrice*). Now suppose the user decides to delete the last predicate on Order and update the unitPrice in Product. Then the addition to $\mathbb{L}$ is (disappears circle *Order.shipCountry* circle *Product.unitPrice*). Note that there is no appears symbol preceding circle *Product.unitPrice* as it is an update of an existing predicate instead of creation of a new one.

Shape descriptors. Using the alphabet, we can define the elementary and derived shapes in an LCG along with their positions on a GUI using shape descriptors. Internally, an LCG sequence is transformed to a sequence of shape descriptors. The syntax for defining a shape descriptor is: (shape *name*(*parameters*) *descriptor*). A shape definition is identified by a *name*, which is followed by a possibly empty list of parameters, and then a descriptor for the shape. For example, here is a definition of a circle: (shape circle(*color*, r, x, y, s) appears circle *Supplier*) where r is the radius of the circle, (x, y) is the position of the center of the circle (*e.g.,* on the *Query Canvas*), s (optional) specifies i-th (*e.g.,* second) occurrence of Supplier in the LCG, and the *color* parameter is used to set the color of a circle. An example definition of a comp_arrow shape descriptor is as follows: (shape comp_arrow(D_n) join appears arrow *ORDERS* appears circle *ORDERS.orderID* appears circle *ORDERS.unitPrice*) where D_n is a set of parameters containing the content of properties *ORDERS.orderID* and *ORDERS.unitPrice*. Note that the former is an example of an elementary shape whereas the latter is a derived shape (recall from Section 3.2). The sizes and positions of the shapes and colors are computed automatically. The meaning of the descriptor will be clear momentarily.

The set of shape descriptors natively supported for LCG construction is given in Table 3. The simplest shape descriptor is of an elementary shape type. The symbols circle, arrow, bold, and empty of the alphabet correspond to same-name shape descriptors of elementary shapes. For example, in Figure 2, the Supplier and SUPPLIES can be defined using the circle and arrow shape descriptors, respectively. Note that empty yields an empty shape (*e.g.,* in response to deletion of a node or link). Derived shapes are defined by recursively combining elementary and previously defined shapes. In

particular, there are four types of derived shape descriptors, namely, *composite circle*, *composite arrow*, *expand*, and *shrink*. Intuitively, *composite circle* (resp. *composite arrow*) represents the derived shape type comprising of a circle (resp. arrow) representing a node (resp. link) and one or more predicate circles (*i.e.,* composite circle and arrow in LCG data model). For example, the Order and ORDERS can be defined using the composite circle and composite arrow shape descriptors, respectively. Suppose now one of the predicate represented by the yellow predicate circle in Order is deleted by the user during query formulation (*i.e.,* disappears circle *Order.shipCountry* in an LCG sequence). This results in decrease in radius of the corresponding predicate circle as well as the Order circle. This modified geometry of Order is represented by the shrink shape descriptor. Alternatively, addition of a predicate circle increases the radius/width of a composite circle/arrow and is represented by expand. Note that we classify these two descriptors as derived since they are only valid for composite circles/arrows.

Visual shape operators. The shape descriptors define the color-coded elementary and derived shapes from an LCG sequence but they do not construct them. Observe that the descriptor of a derived shape begins with *join*, *up*, or *down* which are *visual shape operators*. These operators are used to construct the specific shape instances in an LCG based on the shape descriptors. In particular, they implement algorithms to ensure that the constructed LCG is grounded on the aforementioned theories and principles of HCI, visualization and psychology. We first introduce these operators here and discuss their implementation in the next section.

The creation of elementary shapes is straightforward. Each elementary shape invokes an operator that creates the corresponding object on the query canvas. Specifically, circle (resp. arrow) invokes creation of a circle (resp. arrow) object with predefined name, color, radius (resp. width) and location. The bold shape descriptor invokes an operator to bold the circumference of an existing circle. Empty simply invokes deletion operator to remove selected node/link.

The creation of the derived shapes is more involved. This is primarily because the geometry and positioning of circles in them are automatically generated and maintained. To this end, we introduce the following set of visual shape operators.

Join. We can generate derived shapes using the *join* operation[2]. Specifically, there are two types of join supported in VEDA. The *self join* operator, denoted by $P_i \triangleright P_j$, *joins* two predicate circles on the *same* node/relationship property (*e.g.,* predicates on the shipCountry property) and returns a predicate circle with a larger radius (by invoking the *up* operator). Specifically, since it increases the size of the predicate circle instead of creating a new one for *only* those with multiple predicates, it moderates the visual complexity of an LCG due to quantity of objects. The *join* operator, $S_i \bowtie S_j$, where one of the operand is either a parent circle (arrow) or a derived shape and the other is a predicate circle, returns a composite circle or arrow that combines S_i and S_j. Specifically, it systematically lays a predicate circle S_j on the circumference or length of S_i and maintains it. Hence, it realizes the Gestalt's principles and moderates the impact of visual complexity w.r.t. quantity of information (only necessary predicate circles are created), variety of visual form (same size) and spatial organization (sufficient gaps between them to handle visual congestion). Note that S_i and S_j must be associated with the same node or relationship.

In general, the descriptor of a composite_circle() (resp. composite_arrow()) generates a derived shape $O \bowtie P_1 \; op \; P_2 \; op \; \ldots \; op \; P_n$ where $op \in \{\triangleright, \bowtie\}$, O is a parent circle (resp. arrow) and $P_1, P_2 \ldots P_n$ are predicate circles on the properties of O. For example, Figures 4(b) - (d) can be expressed as $S_1 = O \bowtie P_1$, $S_2 = S_1 \bowtie P_2$, and $S_3 = S_2 \bowtie P_3$, respectively. Assume the new predicate related to

[2]We refer to it as 'join' since at a conceptual level this operation combines two shapes at an intersecting segment (*i.e.,* circumference or length).

the brown predicate circle is denoted as P_4. Then the brown predicate circle in Figure 4(e) can be expressed as $P_1' = P_1 \triangleright P_4$.

Up. The *up* operator, denoted by $\triangle S$, increases the size of a shape S. It is invoked by the join operation to increase the size of the resultant shape. Hence, the descriptor of expand(), (up O P_n), where O is the circle or arrow (elementary shape) and P_n is the predicate circle of O associated with property n, triggers the increase in size of the derived shape. Observe that this results in an increase in the radius of P_n if there already exists at least one predicate on n and an increase in the radius or width of O.

Down. Conversely, the *down* operator, denoted by $\triangledown S$, decreases the size of a shape S with the deletion of an existing P_n in O. Note that while $\triangle$ is implicitly invoked during a join operation, $\triangledown$ is invoked in response to an explicit user action of predicate deletion.

Observe that the *up* and *down* operators realize the principle of importance ordering and controls the visual complexity w.r.t. size variety. In summary, the visual shape operators uphold the Gestalt's principles, moderate the visual complexity of an LCG, reduce the short-term memory load due to color coded predicate circles, and upholds the expressiveness and effectiveness criteria of visualization since they only construct relevant components in an LCG by exploiting shape, position and color.

Remark. Unlike traditional database operators, we do not propose a set of transformation rules to rewrite an expression containing these operators into an equivalent but a more efficient form. This is because our focus is on usability, perceivability and aesthetics of an LCG and not efficient generation of the visual abstraction. Furthermore, during visual query formulation, each action taken by a user must be processed *immediately* so that the GUI can provide informative feedback of the action and support design of dialogue to yield closure (*i.e.,* golden rules of Shneiderman). Hence, it is impractical to defer processing of these actions in order to transform an expression to a more efficient form.

4 ALGORITHMS FOR DERIVED OPERATORS

We now describe the algorithms for implementing the join, up, and down operators. For ease of exposition, we assume a property graph query is specified incrementally in visual querying mode and use the LCG in Figure 2 as the running example.

Join and self join operators. Intuitively, the join operation overlays the predicate circles on the circumference of a parent circle or on the length of an arrow. Specifically, we want to ensure that the predicate circles are *positioned* and *maintained* judiciously so that the aforementioned principles and theories are embraced. We begin with the join algorithm which increases the radius (resp. length) of a parent circle (resp. arrow) without changing the radius of predicate circles. Then we discuss the self join operation which increases the radius of the involved predicate circle. These two operators invoke the $\triangle$ operator, which is discussed later.

Join involving node properties. We assume the *initial radius* of circles representing nodes and predicates are initialized to R and r, respectively, as these values depend on the screen size. Consider the Customer node (denoted by O_{cust}). Suppose the first predicate p_1 is constructed on the country property of Customer. We denote the corresponding predicate circle representing p_1 as P_1. Since P_1 is the first predicate circle of O_{cust}, $O_{cust} \bowtie P_1$ operation randomly positions it on the circumference of O_{cust} but excludes the positions that contain the "hooks" for drawing an arrow (*i.e.,* the small black circles). This is to avoid visual congestion in these regions due to the possibility of a predicate circle eclipsing these hooks. The key issue now is *how to position the subsequent predicate circles?*

Let k_{cust} be the number of distinct properties associated with Customer in the underlying property graph. Then O_{cust} can have at most k_{cust} predicate circles based on our LCG design.

These circles should be positioned symmetrically along the circumference by maintaining sufficient *"gap"* between them. Consequently, the center positions of subsequent predicate circles and the gap between them can be computed automatically by exploiting R, r, k_{cust}, and circle geometry where we assume each hook position can be conceptually represented by a "pseudo" predicate circle [36]. For instance, the Customer node has predicates on eight properties. Observe that the corresponding predicate circles of O_{cust} are evenly spaced, avoiding the hooks. We note that although the number of properties may be large, a user will typically impose predicates on a small subset of them. Furthermore, with the addition of each predicate circle, the radius of the parent circle increases, which in turn maintains adequate gap between the predicate circles. This is visually evident in O_{cust} where even though $k_{cust} = 8$, its radius is increased and adequate gap is maintained between the predicate circles which are positioned symmetrically.

Join involving relationship properties. Next, we describe the join operation involving edges (*i.e.,* arrow) in an LCG. Unlike nodes, the length and position of an arrow are not fixed apriori and are determined by the way a user draws an edge. Hence, we can flexibly assume a predefined gap δ and update an arrow's length ℓ accordingly as new predicate circles are drawn on it. Similar to the query nodes, there can be at most k_n distinct predicate circles on an arrow. These circles should be positioned evenly along the length by maintaining gaps between them. We also wish to avoid the neighborhoods around two end positions. The positions of the predicate circles and the updated length of the arrow can be computed by exploiting k_n, r, ℓ and δ [36]. For example, the two predicate circles on ORDERS are positioned symmetrically with sufficient gap between them and the end points. Note that the positions of all other existing shapes may be updated after the join since the center positions of the source or target nodes may change.

Observe that the above procedures uphold the laws of proximity, symmetry and closure of Gestalt's principles. The symmetrical layout and consistent size of the predicate circles embrace the visual complexity principles of spatial organization and perceivability of details as well as moderate the impact of variety of visual form. This in turn positively impact the aesthetic of the LCG (Berlyne's aesthetic theory).

Self join operation. The self join operation $P_i \triangleright P_j$ simply increases the radius of the existing predicate circle P_i. Hence, this operator can be realized by the $\triangle$ operator.

Up and down operators. Intuitively, these operators enable to visually highlight the "importance" of a query node or edge in an LCG. The larger the number of predicates a node or an edge has the more is its relative importance in an LCG and hence its size is increased accordingly to gain a user's attention. Hence, the implementation of these operators embrace the principles of importance ordering. Recall that the *Up* operator is invoked during the join and self join operations. For the former operation, when a new predicate circle P_i is joined with a (composite) circle (resp. arrow) O_n, the radius (resp. width) of the parent circle (resp. arrow) is first increased by a pre-defined value Δ (*e.g.,* $R + \Delta$) *before* the join is performed. Observe that the increase in width of an arrow does not impact the positions of the predicate circles. Hence, when O_n is a (composite) arrow it simply increases its width by Δ. On the other hand, when O_n is a composite circle, increase in the radius of the parent circle demands update of the positions of existing predicate circles in O_n. Hence these positions are updated [36] and then $O \bowtie P_i$ is performed to generate the new shape. For the latter case of self join, we need to increase the radius of an existing P_i. This is achieved by ensuring the gap between the larger P_i and its neighboring predicate circles remain the same (*i.e.,* prior to the increase in size of P_i) [36]. Otherwise, we may encounter visual congestion (*e.g.,* Figure 4(e)). For instance, consider the yellow predicate circle in Order. It captures three predicates on shipCountry and consequently the *up* operator increases its radius and the radius of Order while maintaining adequate gap with other predicate circles.

Conversely, the *down* operator is invoked when a predicate circle is deleted from a composite circle/arrow or removal of a predicate from an existing multi-predicate predicate circle. It simply reduces the radius (resp. width) of the parent circle (resp. arrow) by Δ or reduces the radius of a predicate circle and update the positions of existing predicate circles in a composite circle using the aforementioned strategy. Note that in case a predicate circle is removed from a composite circle/arrow we do not shift the remaining predicate circles but leave the position empty. Any subsequent addition of a predicate circle can then occupy the position of the deleted circle.

Stitching them together. For a join operation, first the position of the (composite) circle or arrow (*e.g.,* on the query canvas) is retrieved. Then, the radius (resp. width) of the parent circle (resp. arrow) and positions of the existing predicate circles are updated using the *Up* operator. Next, the position of the new predicate circle P on the shape O_n is computed. Finally, the derived shape is rendered on the GUI by updating O_n.

5 AN LCG-BASED VISUAL QUERY INTERFACE

In this section, we begin by reporting our user engagement with a popular commercial GUI to understand what query features are expected to be supported by a visual query interface for property graphs in practice. Next, we present a novel LCG-based GUI for read-only *Cypher* queries called SIERRA to address them.

5.1 BLOOM: Feedback from End Users

Neo4j's Bloom *"is a data exploration tool that visualizes data in the graph and allows users to navigate and query the data without any query language or programming"* [9]. Specifically, its *search with type-ahead suggestions* (*i.e., search suggestions*) enables growing a linear query one edge-at-a-time. For Bloom version 1.4.5, a filter feature allows limited specification of predicates on the search results. Specifically, it does not allow fine-grained imposition of *different* predicates on the *same* node label.

We surveyed 17 unpaid volunteers (ages from 20 to 39), none of them are authors of this paper. These volunteers are students, industry professionals, or researchers. They displayed a range of familiarity and expertise with graph-structured data according to a pre-study survey. Specifically, they are familiar with (or use) graph-structured data in biology, finance, social and commercial products domains. Hence, they understand linear and non-linear graph structures. Several are also familiar with visual graph querying systems and have used them before. Although several volunteers have used programming languages like Java to manipulate graphs or are familiar with SQL, none of them are familiar with the syntax of *Cypher*. After consenting to have their feedback used for research we first requested them to peruse the Bloom website [8] to understand how to visually formulate *Cypher* queries. We then asked them to choose any datasets of their choice in Neo4j and formulate at least 5 queries of their choice using the type-ahead suggestions mode. Next, we ask them a set of questions through a survey related to desirable features for visual formulation of read-only *Cypher* queries and how easy it is for them to represent these features in Bloom. Specifically, there are five questions that map to the five features shown as legends in Figure 5 (b). There are additional four questions related to usability and aesthetics that map to the four items in Figure 5 (c).

Figures 5(b)-(c) depict the results of desirable features of a GUI for *Cypher* queries. A majority of the volunteers thinks that the visual interface should support non-linear structure queries, multiple predicates on nodes and edges, easy visualization of node labels properties and predicates. It should be easy to use and aesthetically pleasing. We also asked them how easy it is to formulate these features in Bloom. Over 76% mentioned that they struggled to visually formulate predicates and

non-linear structure queries in Bloom. Observe that the former is inherently a limitation of the LAG abstraction as highlighted in Section 1.

5.2 SIERRA

Visual actions. We begin by formally defining *visual actions* (*actions* for brevity), which are GUI-level actions taken by users to formulate read-only *Cypher* queries. These actions can be mapped to the symbols in VEDA alphabet internally (3rd col. in Table 2). Specifically, these actions are:

- add(q,c): The add action denotes a user adding a query component (edge/link, node, predicate) c to an existing query graph q (possibly empty) and returns the augmented query;
- delete(q,c): This action denotes that a user revokes (deletes) a query component c and returns the modified query graph. A deletion of a node or link triggers deletion of all predicates associated with it and any dangling links;
- update(q,c): This action denotes that a user modifies an existing query component c, and returns the modified query graph. The set of modifications includes update of an existing predicate and update of return nodes. Update of a node/link label is modelled by a sequence of delete(·) and add(·) actions;
- return(q,c): This action enables a user to specify the node c to be returned in the results of q.

Note that we do not model low-level operations (*e.g.,* mouse click, drag-and-drop) as visual actions as different GUIs typically follow different sequences of these operations to realize these four visual actions.

We refer to a sequence of visual actions taken by a user to formulate a query as *action sequence* which is transformed to the corresponding LCG sequence in VEDA. We assume the query is a valid *Cypher* query. It is easy to observe that the same query can be constructed by different action sequences. Since at query time a user specifies a particular action sequence, it can be unambiguously transformed to the corresponding LCG sequence.

Preprocessing property graphs. Given a property graph, we first preprocess it offline by extracting distinct node labels and relationship types and properties associated with them. For each node label, we record the set of node labels connected to it. We store them in a set of indexes for efficient access.

SIERRA GUI. Figure 2 depicts the SIERRA GUI. The left panel displays (on demand) the list of node labels, relationship types and corresponding properties. The middle panel enables a user to construct a read-only *Cypher* query visually using direct manipulation [57]. The right panel shows the corresponding textual *Cypher* query of the (partially) constructed visual query in real time. We focus on how the aforementioned actions are realized in SIERRA to generate an LCG-based visual representation of a query. Since the visual representation is based on LCG, the GUI design is informed by aforementioned theories and principles of HCI, visualization and psychology as summarized in Table 1. Note that we do not focus on query result visualization in the sequel as it is orthogonal to the problem addressed in this work.

Implementation of add(·): A user can create a labeled or unlabeled query node easily in SIERRA. On hovering over each node label option, two icons appear on the right side of the option, an *eye* and a *plus* icons, representing View Properties and Add Node, respectively. On clicking the *eye* icon, one can view a list of properties associated with the node label. On clicking the *plus* icon, one can add a corresponding query node in the *Query Canvas* (*i.e.,* invokes the circle operator in VEDA). The node is displayed as a labeled circle and a color is assigned to it dynamically by matching it with the color of the corresponding node label. On the other hand, if a user wishes to create an unlabeled query node, she can simply right click on an empty space in the *Query Canvas* and

choose Add Node option from a drop-down menu to create an unlabeled node. Such a node is not given any color code.

When one clicks on a labeled query node, a sidebar pops up on the left of the screen, showing a list of properties that can be considered for predicates as well as a list of labels of possible *targets*. These targets are node labels that are connected to the query node label in the underlying property graph. We can add a target node on the *Query Canvas* in two ways. First, one may click on the *plus* icon that appears when the mouse is hovered over a target (*e.g.,* Product target of Order). Alternatively, one may add a target node in the same way as we create a labeled or unlabeled query node.

After a node pair is added, we can then draw an edge (shown as arrow) by dragging the cursor from source to target node (invokes the arrow operator). Clicking on the edge displays in the left panel a dropdown list of all possible relationship types between these node pairs. One can optionally select a relevant type (*e.g.,* ORDERS) as well as any distance bounds for variable length patterns. In the case, the underlying property graph does not have an edge between the two node labels, SIERRA warns the user. A user can acknowledge it and continue constructing the edge if necessary.

One can simply click on a query node (resp. edge) and select a property from the displayed property list on the left panel to add a predicate. A predicate circle appears along the circumference (resp. length) of the node (resp. edge) to visually represent it in the form of composite circle (resp. arrow). This circle is filled with the same colour as the property color, generated dynamically, allowing a user to easily associate it with its property (Figure 2). Next, one can fill in the operator and value associated with the predicate by selecting from drop-down lists. One can add multiple predicates on a single property or on multiple properties of a node or relationship (invokes the join or self join operators). Recall that the positioning and size of a predicate circle and query node/link are automatically maintained by VEDA (invokes the join and up operators).

Implementation of update(·): A user can click on a node, edge, or predicate circle to update the content in consistent with above definition of the visual action.

Implementation of delete(·): To delete any node or edge or predicate, one can simply select the component by clicking on it, followed by pressing the backspace or delete key. Under the hood, this results in invocation of the down or empty operators.

Implementation of return(·): Lastly, a user can mark the node that should be returned in the results by simply double clicking on the corresponding query node and confirming it to be a return node through a dialogue box. A return node is depicted with a bold circle. Under the hood, the bold operator in VEDA is invoked.

Remark. Observe that SIERRA is orthogonal to the underlying property graph query engine. A visually formulated query in SIERRA can be transformed to the corresponding *Cypher* query and executed by any state-of-the-art query engine.

6 USER STUDY

SIERRA is implemented with React. In this section, we investigate the performance of SIERRA and report the key findings. All experiments are performed on a 64-bit Windows desktop with Intel(R) Xeon(R) W-2235 CPU (3.80GHz) and 32GB of main memory.

6.1 Experimental Setup

Datasets. We use the following datasets that are built-in for the *Neo4j Browser*: (a) The *Movies* dataset consists of movies and people that are related (*e.g.,* director, producer, actor) to these movies. The *Movies* property graph contains 171 vertices (*i.e.,* 38 movies and 133 people) and 253 edges

Table 4. Summary of tasks.

Task Id	GUI	# of nodes	# of edges	# of predicates	Dataset
1	Bloom, SIERRA	3	2	1	Northwind
2	Bloom, SIERRA	4	3	2	Northwind
3	Bloom, SIERRA	4	3	1	Movies
4	Bloom, SIERRA	6	5	2	Northwind
5	SIERRA	3	3	10	Movies
6	SIERRA	4	4	12	Northwind
7	SIERRA	5	5	10	Movies
8	SIERRA	5	5	28	Northwind
9	SIERRA	6	6	13	Northwind
10	SIERRA	5	4	19	Northwind

(*i.e.,* 172 acted-in, 44 directed, 15 produced, 10 wrote, 3 follows, 9 reviewed relationships). (b) The *Northwind* dataset contains sales data involving imports and exports of specialty foods. The *Northwind* property graph contains 300 vertices (*i.e.,* 77 products, 8 categories, 29 suppliers, 91 customers and 95 orders) and 504 edges (*i.e.,* 77 part-of, 77 supplies, 95 purchased and 255 orders relationship). Note that since our focus is on visual abstraction of property graph queries, the size of the underlying property graph is orthogonal to the problem.

Baselines. We compare SIERRA with the following baselines.

- Bloom: We use Bloom [9] as a representative of the LAG abstraction. We focus on the search suggestions option in Bloom (Section 5.1). Note that VISAGE [47, 48] is not publicly available. We also do not use *Graphistry* [6] as it is in Beta phase and does not support the above datasets.
- LAG-SIERRA: We create a variant of SIERRA to compare the LCG abstraction with its LAG counterpart. To this end, we disabled creation of predicate circles and all nodes have identical shapes. The links are not maintained either. A user may double-click on any query node or link to view the set of predicates associated with its properties.
- Cnt-SIERRA: We create another variant of SIERRA to investigate the impact of different-sized predicate circles (Figure 6 (left)). Instead of showing predicate(s) on each property as a predicate circle, we display a red circle with a number in it, indicating how many properties have predicates on a query node/link. That is, a user can view the total number of properties involved in predicates but is unable to distinguish predicate(s) on each property.

Participants. 20 unpaid volunteers, within the age group of 21 to 39 years old, participated in the study in accordance to HCI research that recommends at least 10 participants [26, 33]. All participants indicated an interest in working with graph databases. Our participants come from different professional backgrounds, including computer science and engineering students, senior software engineers, business analysts and product managers. Six of the participants indicated that they have worked with graph databases before. We informed the participants about the purpose of study. After consenting to have their feedback used for research, we gave them a walkthrough of how visual queries can be formulated using SIERRA (and the two variants) and Bloom. We allowed them to explore these two interfaces and answered any queries related to them. Once done, we gave them a series of 5 tasks to complete.

Tasks and procedure. A key goal of our user study is to experimentally evaluate the benefits of the novel features introduced by the proposed LCG abstraction compared to its LAG counterpart. Does these features bring benefits to the users in terms of usability, perceivability and aesthetics? Hence, it is important for the participants to undertake tasks that involve experiencing these two abstractions. To this end, we involve them to construct a set of visual property graph queries on Bloom and SIERRA (and its two variants). We provided printed visual queries to the participants. A subject then constructs the given queries visually using a mouse. They were informed to undertake

Table 5. LCG Features (No. of participants vs rating scores).

ID	Item focused by survey questions	*1*	*2*	*3*	*4*	*5*
Q1	The differences in node/predicate sizes help in query construction.	0	0	3	8	9
Q2	The color coding of the nodes are useful.	1	1	2	8	8
Q3	The color coding of the predicate circles are useful.	1	1	2	8	8
Q4	The predicates are well positioned around the nodes.	0	0	0	5	14
Q5	The predicates are well positioned around the edges.	0	0	0	6	14
Q6	The predicate circles provides useful information.	0	2	4	8	6
Q7	The predicate circles are distracting.	16	4	0	0	0
Q8	The LCG diagram looks aesthetically pleasing.	0	0	2	7	11

their tasks at leisure. Once the participants have completed their assigned tasks, each of them filled up a survey form with a set of questions and engaged in an interview about their experience.

There are 10 tasks in total (Table 4). Each task involves formulation of a property graph query visually. These tasks are chosen based on varying complexities in terms of topology, number of nodes, links, and predicates. Since Bloom does not support visual construction of non-linear graph queries, we divide the tasks into two pools, *Tasks 1* to *4* are linear queries and *Tasks 5 – 10* are non-linear queries. To prevent fatigue among our participants, we select 2 tasks from the first pool and 3 tasks from the second one for each of them. We ask the participants to undertake the tasks in all three variants of SIERRA and Bloom (if possible). At the end of the user study process, each task was performed at least 5 times by the volunteers. Moreover, to reduce bias, we randomize the sequence of the four GUIs used by the participants. The order in which the tasks are presented to participants is also randomized.

6.2 Features of LCG

In our survey to the participants, we asked questions related to the features of LCG. Each subject gave a rating in the Likert scale of 1-5 (1 = strongly disagree and 5 = strongly agree). Table 5 reports the results. Note that the survey is taken by the participants after they finished the assigned tasks using the four GUIs. Hence, they experienced the LAG abstraction (Bloom and LAG-SIERRA), LCG abstraction (SIERRA), and "semi"-LCG abstraction (Cnt-SIERRA) before giving feedbacks. We can make the following observations. First, the dynamic maintenance of size of various components using VEDA is agreeable to 85% of the subjects even after using Cnt-SIERRA ($Q1$). It justifies the effectiveness of area as a perceptual task in visualization. Second, the idea of predicate circles (*i.e.,* derived shapes) and their careful positioning have positive impact on the users. None find them distracting ($Q7$). All except one (this subject did not provide feedback on $Q4$) find the predicates are well-positioned ($Q4$, $Q5$). Overall, more than 70% find the predicate circles provide useful information ($Q6$) even after experiencing the LAG abstraction. Third, while at least 70% find color coding of nodes and predicate circles helpful, 10% disagrees ($Q2$, $Q3$). We believe this issue can be easily addressed by personalizing the color coding feature in a GUI - enabling a user to disable it, if necessary. Lastly, majority of the subjects find the LCG view of a query aesthetically pleasing ($Q8$). This is consistent with Berlyn's aesthetic theory as moderate degree of visual complexity is desirable for aesthetically pleasing and usable GUI. Hence, higher visual complexity of the LCG abstraction compared to LAG does not adversely impact aesthetics and usability.

6.3 Subjective Feedback

We asked participants on their experience in using the GUIs of SIERRA and Bloom and provide unstructured feedback. Based on the feedback and also our observations of the query formulation sessions, participants appear to enjoy using SIERRA. 75% of subjects rated SIERRA superior to

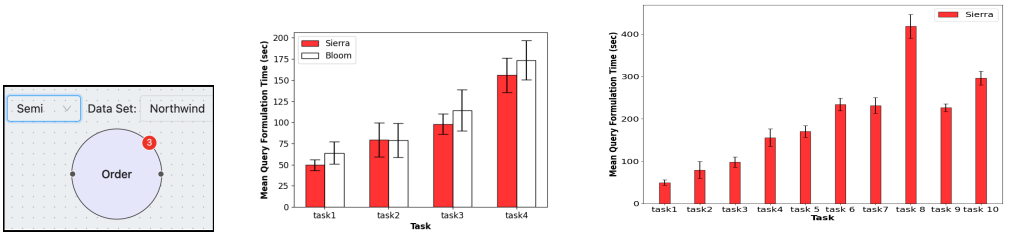

Fig. 6. Cnt-SIERRA interface (right); mean query formulation times (middle, right).

Bloom in conveying query-related information visually. This is intuitive as SIERRA leverages color coding, predicate circles, and size-varying shapes judiciously to convey information at a glance.

One participant, for instance, found SIERRA more responsive to his/her input during query formulation: *"SIERRA is more responsive than Bloom, which (query suggestions) takes too long to load."* Another participant found the LCG-based query formulation in SIERRA very intuitive: *"It was pretty easy to begin carrying out the tasks with just a short introduction on its functions. The query construction process was very intuitive."* One commented on the aesthetic aspect of the LCG-based visualization of a property graph query: *"Looks very nice and bright."* Another emphasized on the informativeness of an LCG-based GUI in comparison to LAG-based one: *"Easy to use, provides more information than Bloom."* One subject observed the limitations of Bloom in adding predicates: *"For Bloom, hard to write the predicate when using the autocorrect."*

Observe that in our proposed abstraction we do not explicit add the details of predicates associated with a predicate circle in order to avoid visual clutter. This feature along with the interaction associated with it in SIERRA was mentioned by a participant: *"The details of predicates were not shown to and I have to click on the node and its various properties before observing the value of the properties."*

6.4 Query Formulation Times (QFT)

The above results demonstrate the superiority of the LCG abstraction to its LAG counterpart. Lastly, we investigate whether an LCG-based GUI can support efficient construction of queries compared to an existing industrial-strength LAG-based GUI. To this end, we compare the QFT in SIERRA and Bloom. We note that Bloom has limited features and different interface design. Hence, it is challenging to undertake a comparative study to focus only on the "LCG effect" without redesigning it. Importantly, in practice, several factors related to a GUI (*e.g.,* interface design, visual abstraction, efficient implementation) impact the QFT. Hence, in this study our goal is to compare QFT and *not just* the "LCG effect".

We record the time taken to formulate a query (*i.e.,* task) by the subjects. Figure 6 (middle) plots the results of *Tasks 1-4*. Observe that the mean QFT of SIERRA is lower than Bloom for 3 out of 4 tasks. We used Welch's t-test to test the null hypothesis that the mean QFT for Bloom is less than or equal to the mean QFT for SIERRA for each of the 4 tasks. The t-test p-values of *Tasks* 1, 2, 3, and 4 are 0.0045, 0.5131, 0.0404, and 0.0422, respectively. Since the p-values of tasks 1, 3, and 4 are all less than 0.05, we can reject the null hypothesis for 3/4 tasks and assert that the mean QFT for Bloom is significantly greater than that of SIERRA. We *hypothesize* (*i.e.,* we do not claim to have evidence yet) based on feedbacks from users (preceding subsections) that the LCG abstraction may contribute to efficient query formulation.

Figure 6 (right) depicts the mean QFTs in SIERRA for all queries. Naturally, the QFT grows with the number of nodes, links, and predicates in a query. Recall that *Tasks* 5-10 cannot be formulated

using Bloom. Hence, SIERRA is efficient and supports the construction of more complex queries compared to Bloom.

Lastly, since efficient query formulation is influenced by the ease with which links and predicates can be added in an LCG, we surveyed the subjects on the ease of adding them in SIERRA. We observe that 80% and 95% participants agreed or strongly agreed that it is easy to add links and predicates, respectively, while formulating a query.

7 RELATED WORK

Visual graph query interfaces have gained increasing attention in recent times [4, 8, 12, 21, 22, 30, 32, 47, 48, 62, 63, 65]. All these efforts represent a visual query graph using the LAG abstraction where a node/edge is typically associated with a *single* label. In contrast to the LCG abstraction, these approaches either do not allow visual formulation of multiple predicates or fail to effectively represent them visually. In addition, these efforts do not systematically exploit principles and theories of HCI and psychology in designing their visual abstractions. That is, they fail to explain how theories and principles permit to take the leap to a specific design solution. The LCG abstraction proposed in this work embodies "counterfactual thinking" [45] where principles and theories of HCI, visualization and psychology inform its design.

Most germane to this work are efforts in *visual query representation* (VQR), which is a visualization to represent a query regardless of the query results [29, 34, 38]. These work focus on SQL queries and represent them using graphs. SQLVis [38] focuses on VQR for supporting query formulation where tables and relations are represented as nodes and edges, respectively. A node includes the name of the corresponding table and an alias (if any). It can be expanded to view the corresponding schema. In contrast to LCG, all nodes have the same shape, size, and color. The edges are labeled (*e.g.,* join conditions, join types). Nested queries are visualized using boxes and color saturation. This representation is guided by Gestalt and visualization principles. QueryVis [29, 34] is a seminal effort that aims to capture the first order logic representation of an SQL query to represent its underlying logic visually. It is specifically designed for query interpretation and not for query formulation. The nodes here abstract a table and their attributes. A bounded box is used over the tables based on the quantifier applied to a node. Attributes involved in predicates are written as new rows in the table. Directed edges between attributes of tables are utilized to represent join predicates. This abstraction is guided by several visualization principles (*e.g.,* minimality, effectiveness [37])). Similar to these efforts, the LCG abstraction is also a graph. However, while the nodes are of same shape, they can be of different sizes and colors. Furthermore, in contrast to these efforts, it visually encodes the predicates in a query using predicate circles and position them judiciously. The LCG also exploits a broader set of theories and principles from HCI, visualization and psychology for guiding its design. These are materialized using a novel language called VEDA.

Composite circles have been exploited by the visualization community to render network properties [17, 43, 64]. For instance, *hybrid layouts* integrate multiple strategies to laying out network topology [43]. In particular, composite circles are used in [17] to integrate Treemap encoding for the hierarchical structure of a network with other network encodings where a layer is represented using a circle. *Graph Thumbnails* [64] exploits circle packing to visualize an overview of hierarchical network structure in order to support rapid browsing of a large collection of graphs. In comparison, composite circles in the LCG abstraction are designed to represent predicates in property graph queries and not for representing hierarchical network structure. Structurally, the predicate circles in an LCG are laid out systematically *only* on the circumference of a parent circle whereas in these work a circle may be subsumed by a larger circle. More importantly, in contrast to these efforts, our design is informed by theories and principles from psychology and HCI.

Oulasvirta and Hornbaek [45] recently lamented that the usage of theory in HCI to inform design or construction is surprisingly rare for various reasons such as belief in the power of iterative design, focus on whether user interfaces work and not why they work or fail, etc. In particular, much functionalities is not justified theoretically (*e.g., "we include this functionality because of [X]"*). In this work, we take a concrete step towards designing a visual abstraction that is informed by principles and theories.

8 CONCLUSIONS

Despite the growing popularity of property graphs, visual frameworks for formulating property graph queries are still in its infancy. In this paper, we present a novel visual abstraction for read-only property graph queries called labeled composite graph (LCG) that departs from the traditional node-link diagram (*i.e.,* LAG abstraction) view. An LCG not only supports richer features compared to its LAG counterpart but also embraces HCI, visualization and psychology principles and theories to inform its design to create more usable and aesthetic query interfaces. To this end, it deploys VEDA, a novel language for defining, constructing and maintaining LCG. We build a novel visual query interface coined SIERRA that embodies the LCG abstraction and demonstrates with a user study its superiority to a classical LAG-based visual querying system.

The work reported in this paper is inspired by the notion of "counterfactual thinking" [45] which essentially advocates for theory-informed design. An interesting direction of future research is its formal treatment (*i.e.,* counterfactual reasoning [45]) in the context of LCG abstraction. Furthermore, in this paper we focused on property graph queries that are formulated by end users. In the case the queries are application-generated, it is possible that they may have many predicates. This opens up the possibility of expanding the set of psychology theories considered in this work by exploring the roles *cognitive load theory* [58] and *color theories* [14] may play in designing superior visual abstractions. In summary, we hope the LCG abstraction will trigger interest in not only visual querying but also the "constructive power" [45] of theories in informing the design of usable and aesthetic visual query interfaces.

ACKNOWLEDGEMENT

Sourav S Bhowmick is partially supported by Ministry of Education (MOE) AcRF Tier 1 Grant No. RG80/21.

REFERENCES

[1] The Aesthetic-Usability Effect. https://www.nngroup.com/articles/aesthetic-usability-effect/ (Last Accessed: Dec 5, 2022).
[2] Cypher for Apache Spark. https://github.com/opencypher/cypher-for-apache-spark/.
[3] Cypher for Gremlin. https://github.com/opencypher/cypher-for-gremlin/.
[4] DrugBank interface. https://go.drugbank.com/structures/search/small_molecule_drugs/structure.
[5] Gestalt psychology. https://en.wikipedia.org/wiki/Gestalt_psychology.
[6] Graphistry. https://www.graphistry.com/ (Last Accessed: Dec 5, 2022).
[7] MemGraph. https://memgraph.com/.
[8] Neo4j Bloom. https://neo4j.com/bloom (Last Accessed: Dec 5, 2022).
[9] Neo4j Bloom User Interface Guide. https://neo4j.com/developer/neo4j-bloom/ (Last Accessed: Dec 5, 2022) .
[10] Cypher Query Language. https://neo4j.com/developer/cypher/.
[11] PGQL Property Graph Query Language https://pgql-lang.org/.
[12] PubChem interface. https://pubchem.ncbi.nlm.nih.gov//edit3/index.html.
[13] RedisGraph. https://oss.redislabs.com/redisgraph/.
[14] Theory of Colours. https://en.wikipedia.org\T1\guilsinglrightwiki\T1\guilsinglrightTheory_of_Colours.
[15] A. V. Aho, J. D. Ullman. Abstractions, their algorithms, and their compilers. *Commun. ACM*, 65(2): 76-91, 2022.
[16] P. Andlinger. Graph DBMS increased their popularity by 500% within the last 2 years. https://db-engines.com/en/blog_post/43, 2015.

[17] D. Archambault, T. Munzner, D. Auber. GrouseFlocks: Steerable Exploration of Graph Hierarchy Space. *IEEE Trans. Vis. Comput. Graph.*, 14(4): 900-913, 2008.

[18] T. Armstrong and B. Detweiler-Bedel. Beauty as an Emotion: the Exhilarating Prospect of Mastering a Challenging World. *Review of General Psychology*, 12, 4, 305-329, 2008.

[19] D. E. Berlyne. Conflict, arousal, and curiosity. *McGraw-Hill*, 1960.

[20] D. Berlyne. Studies in the new Experimental Aesthetics. Washington D.C., *Hemisphere Pub. Corp.*, 1974.

[21] S. S. Bhowmick, K. Huang, H. E. Chua, Z. Yuan, B. Choi, S. Zhou. AURORA: Data-driven Construction of Visual Graph Query Interfaces for Graph Databases. *In SIGMOD*, 2020.

[22] D. H. Chau, C. Faloutsos, et al. Graphite: A visual query system for large graphs. In ICDM, 2008.

[23] W. S. Cleveland, R. McGill. Graphical Perception: Theory, Experimentation and Application to the Development of Graphical Methods. *J. Am. Stat. Assoc.*, 79 (387), 1984.

[24] W. Craig. Gestalt Principles Applied in Design, https://www.webfx.com/blog/web-design/gestalt-principles-applied-in-design/.

[25] L. Deng, M. S. Poole. Affect in Web Interfaces: A Study of the Impacts of Web Page Visual Complexity and Order. *Mis Quarterly*, 34(4), 2010.

[26] L. Faulkner. Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods, Instruments, & Computers*, 35(3), 2003.

[27] A. Forsythe, N. Sheehy, M. Sawey. Measuring Icon Complexity: An Automated Analysis. *Behaviour Research Methods, Instruments, & Computers*, 35(2), 2003.

[28] N. Francis, A. Green, et al. Cypher: An evolving query language for property graphs. *In SIGMOD*, 2018.

[29] W. Gatterbauer, C. Dunne, H. V. Jagadish, M. Riedewald. Principles of Query Visualization. *IEEE Data Engineering Bulletin*, 45(3), September 2022.

[30] F. Haag, S. Lohmann, S. Bold, T. Ertl. Visual SPARQL Querying based on Extended Filter/flow Graphs. *In AVI*, 2014.

[31] S. Harper, C. Jay, E. Michailidou, H. Quan. Analysing the Visual Complexity of Web Pages using Document Structure. *Behaviour & Information Technology*, 32(5), 2013.

[32] N. Jayaram, S. Goyal, C. Li. VIIQ: Auto-Suggestion Enabled Visual Interface for Interactive Graph Query Formulation. *In PVLDB*, 8(12), 2015.

[33] J. Lazar, J.H. Feng, H. Hochheiser. Research Methods in Human-Computer Interaction. John Wiley & Sons, 2010.

[34] A. Leventidis, J. Zhang, C. Dunne, W. Gatterbauer, H. V. Jagadish, M. Riedewald. QueryVis: Logic-based Diagrams help Users Understand Complicated SQL Queries Faster. *In SIGMOD*, 2020.

[35] P. Liu, Z. Li. Task complexity: A review and conceptualization framework. *Int. J. Ind. Ergon.* 42, 2012.

[36] J. Ma, S. S. Bhowmick, B. Choi, L. Tay. Theories and Principles Matter: Towards Visually Appealing and Effective Abstraction of Property Graph Queries. *Technical Report*, https://personal.ntu.edu.sg/assourav/TechReports/SIERRA-TR.pdf, 2022.

[37] J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Trans. on Graphics*, 5(2), 1986.

[38] D. Miedema, G. Fletcher. SQLVis: Visual Query Representations for Supporting SQL Learners. *In VL/HCC*, 2021.

[39] A. Miniukovich, A. De Angeli. Quantification of Interface Visual Complexity. *In AVI*, 2014.

[40] A. Miniukovich, A. De Angeli. Computation of Interface Aesthetics. *In CHI*, 2015.

[41] A. Miniukovich, S. Sulpizio, A. De Angeli. Visual complexity of graphical user interfaces. *In AVI*, 2018.

[42] M. Nadal, E. Munar, G. Marty, C. J. Cela-Conde. Visual Complexity and Beauty Appreciation: Explaining the Divergence of Results. *Empirical Studies of the Arts*, 28(1), 2010.

[43] C. Nobre, M. D. Meyer, M. Streit, A. Lex. The State of the Art in Visualizing Multivariate Networks. *Comput. Graph. Forum*, 38(3): 807-832, 2019.

[44] A. Oliva, M. L. Mack, M. Shrestha, A. Peeper. Identifying the Perceptual Dimensions of Visual Complexity of Scenes. *In Proc. of the 26th Annual Meeting of the Cognitive Sc. Society*, 2004.

[45] A. Oulasvirta, K. Hornbaek. Counterfactual Thinking: What Theories Do in Design. *Int. Journal of Human-Computer Interaction,* 38(1), 2022.

[46] M. Paradies. Graph pattern matching in SAP HANA. *First openCypher Implementers Meeting*, Feb. 2017. https://tinyurl.com/ycxu54pr.

[47] R. Pienta, A. Tamersoy, A. Endert, S. Navathe, H. Tong, D. H. Chau. VISAGE: Interactive Visual Graph Querying. In *AVI*, 2016.

[48] R. Pienta, F. Hohman, et al. Visual graph query construction and refinement. *In SIGMOD*, 2017.

[49] R. Pienta, F. Hohman, et al. VIGOR: Interactive visual exploration of graph query results. *IEEE Trans. Vis. Comput. Graph.* 24(1), 2018.

[50] R. Pieters, M. Wedel, R. Batra. The Stopping Power of Advertising: Measures and Effects of Visual Complexity. *Journal of Marketing*, 74(5), 2010.

[51] D. Pham, S. S. Bhowmick. VOYAGER: Automatic Computation of Visual Complexity and Aesthetics of Graph Query Interfaces. *In EDBT*, 2023.
[52] R. Reber. Processing Fluency, Aesthetic Pleasure, and Culturally Shared Taste. *In Aesthetic Science: Connecting Minds, Brains, and Experience*, 2012.
[53] R. Reber, N. Schwarz, P. Winkielman. Processing Fluency and Aesthetic Pleasure: is Beauty in the Perceiver's Processing Experience? *Personality and Social Psychology Review*, 8, 4, 2004.
[54] A. S. Reber. Gestalt psychology. *The Penguin Dictionary of Psychology*, Viking, ISBN 9780670801367, 1985.
[55] R. Rosenholtz, Y. Li, L. Nakano. Measuring Visual Clutter. *Journal of vision*, 7(2), 2007.
[56] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4), 420-431, 2017.
[57] B. Shneiderman, C. Plaisant. Desigining the user interface: Strategies for effective human-computer interaction. 5th Ed., Addison-Wesley, 2010.
[58] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science,* 12(2):257–285, 1988.
[59] A. N. Tuch, E. E. Presslaber, M. Stocklin, K. Opwis, J. A. Bargas-Avila. The Role of Visual Complexity and Prototypicality Regarding First Impression of Websites: Working Towards Understanding Aesthetic Judgments. *International Journal of Human-Computer Studies*, 70, 2012.
[60] J. M. Wolfe. Guided Search 2.0: A Revised Model of Visual Search. *Psychon Bull Rev*, 1: 202–238, 1994.
[61] E. Wong. Shneiderman's Eight Golden Rules Will Help You Design Better Interfaces. https://www.interaction-design.org/literature/article/shneiderman-s-eight-golden-rules-will-help-you-design-better-interfaces, 2021.
[62] S. Yang, Y. Xie, Y. Wu, T. Wu, H. Sun, J. Wu, X. Yan. SLQ: A User-friendly Graph Querying System. *In SIGMOD*, 2014.
[63] P. Yi, B. Choi, S. S. Bhowmick, J. Xu. AutoG: A Visual Query Autocompletion Framework for Graph Databases. *In The VLDB Journal*, 2017.
[64] V. Yoghourdjian, T. Dwyer, K. Klein, K. Marriott, M. Wybrow. Graph Thumbnails: Identifying and Comparing Multiple Graphs at a Glance. *IEEE Trans. Vis. Comput. Graph.*, 24(12): 3081-3095, 2018.
[65] Z. Yuan, et al. Towards Plug-and-play Visual Graph Query Interfaces: Data-driven Canned Pattern Selection for Large Networks. *PVLDB*, 14(11), 2021.

Received October 2022; revised January 2023; accepted February 2023