Data-driven Visual Graph Query Interface Construction and Maintenance: Challenges and Opportunities

Sourav S Bhowmick
School of Computer Science and Engineering
Nanyang Technological University
Singapore
assourav@ntu.edu.sg

Byron Choi
Department of Computer Science
Hong Kong Baptist University
Hong Kong
bchoi@comp.hkbu.edu.hk

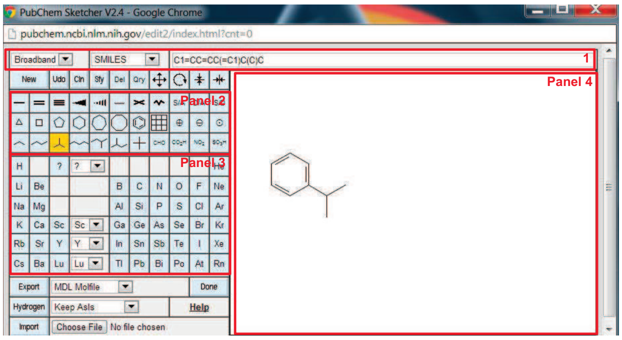
Curtis Dyreson
Department of Computer Science
Utah State University
USA
curtis.dyreson@usu.edu

ABSTRACT

Visual query interfaces make it easy for scientists and other non-expert users to query a data collection. Heretofore, visual query interfaces have been statically-constructed, independent of the data. In this paper we outline a vision of a different kind of interface, one that is built (in part) from the data. In our *data-driven* approach, the visual interface is *dynamically* constructed and maintained. A data-driven approach has many benefits such as reducing the cost in constructing and maintaining an interface, superior support for query formulation, and increased portability of the interface. We focus on graph databases, but our approach is applicable to several other kinds of databases such as JSON and XML.

1. INTRODUCTION

Graphs are a natural way of modeling data in a wide variety of domains and have been extensively studied in mathematics and many areas of computer science. Graph data in real-world applications such as biological and chemical databases (*e.g., PubChem*), social networks (*e.g., Twitter*), co-purchase networks (*e.g.,* Amazon.com), and information networks (*e.g., DBpedia*) has lead to a rejuvenation of research on graph data management and analytics. Several novel graph data management platforms have emerged from academia, industrial research labs (*e.g., Trinity*), and startup companies (*e.g., GraphX*). Several database query languages have been proposed for textually querying graph databases, *e.g.,* SPARQL, Cypher[1], and GraphQL [17]. Creating queries in these languages often demands considerable cognitive effort from users and requires "programming" skill that is at least comparable to SQL [2]. A user must be familiar with the syntax of the language, and must be able to express her needs accurately in a syntactically correct form. However, in many real life domains (*e.g.,* life sciences, social science, chemical science) it is unrealistic to assume that end users are proficient in such query languages. For example, chemists cannot be expected to learn the complex syntax of a graph query language in order to formulate meaningful substructure queries over a chemical compound database such as *PubChem*[2] or *eMolecule*[3].


PubChem Sketcher V2.4 - Google Chrome
pubchem.ncbi.nlm.nih.gov/edit2/index.html?cnt=0
1
New
Udo
Cln
Sfy
Del
Qry
Panel 2
Panel 3
Panel 4
Export
Done
Hydrogen
Help
Import
Choose File

Broadband SMILES C1=CC=CC(=C1)C(C)C

Export MDL Molfile

Hydrogen Keep Asis

Import No file chosen

Figure 1: GUI for substructure search in *PubChem*.

A popular approach to make query formulation user-friendly is to provide a visual query interface (a.k.a GUI) for interactively constructing queries. A GUI for graph query formulation is usually composed of several panels, such as a panel to display the set of labels or attributes of nodes or edges of the underlying data graphs, a panel to construct a graph query graphically, a panel containing *canned patterns* to expedite query formulation, and a results panel to view query results. Intuitively, a *canned pattern* is a small topological pattern (*e.g.,* a benzene ring) which users can drag-and-drop into a query. For example, Figure 1 depicts the screenshot of a visual interface provided by *PubChem* for substructure or subgraph search on chemical compounds. Specifically, Panel 3 provides a list of chemical symbols that a user can choose from to assign labels to nodes of a graph query. Panel 2 lists a set of canned patterns which a user may drag-and-drop in Panel 4 during visual query construction. Note that the availability of canned patterns greatly improves the usability of the interface by enabling users to quickly construct a graph query with fewer clicks than when constructed in an "edge-at-a-time" mode. For instance, the query in Panel 4 can be constructed from two such canned patterns in Panel 2 instead of taking the tedious route of constructing nine edges iteratively. Particularly, Panel 2 is useful if (a) there exists a sufficiently diverse collection of canned patterns in Panel 2 that can aid a user to formulate most (if not all) of her queries; and (b) a user can quickly absorb and find relevant patterns from the collection.

[1] http://neo4j.com/docs/stable/cypher-query-lang.html

[2] http://pubchem.ncbi.nlm.nih.gov/

[3] https://www.emolecules.com/

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.

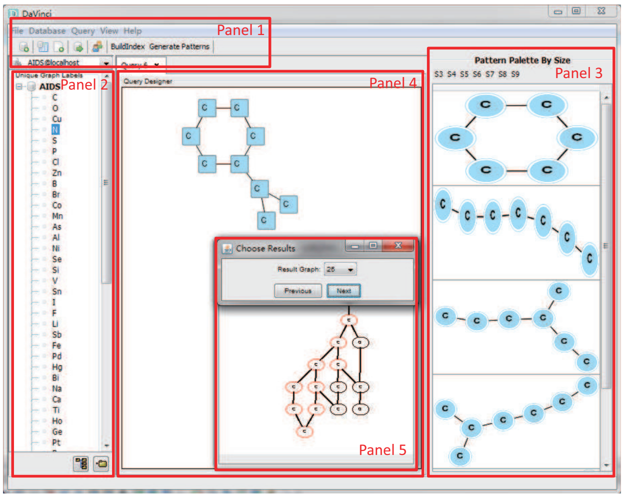


Figure 2: GUI for graph query formulation.

1.1 Data-unaware Visual Query Interfaces

Visual query interfaces utilize the results of decades of research by the HCI community related to various theoretical models of visual tasks, menu design, and human factors. Unfortunately, despite the significant progress this community has made towards constructing user-friendly visual interfaces, three key drawbacks hinder progress.

1. **Lack of diverse content.** First, the content of various components (*e.g.,* Panels 2 and 3) are created manually by "hard coding" them during GUI implementation. Consequently, the set of canned patterns is limited to those selected in advance by a domain expert, *e.g.,* the patterns in Panel 2 are manually selected and added to the GUI. A user may find the pre-selected patterns in Panel 2 useless in formulating some queries. Similar problem also arise in Panel 3 where the labels of nodes are manually added instead of automatically generated from the underlying data.
2. **Static content.** Second, the visual interface is static. That is, the content of Panels 2 and 3 remains static even when the underlying data evolves. As a result, some patterns (resp. labels) in Panel 2 (resp. Panel 3) may become obsolete as graphs containing such patterns (resp. labels) may no longer exist. Similarly, some new patterns (resp. labels), which are not in Panel 2 (resp. Panel 3), may emerge due to the addition of new data graphs or new nodes in existing graphs.
3. **Lack of portability.** Third, classical query interfaces lack portability as a GUI cannot be seamlessly integrated with another graph repository in a different domain (*e.g.,* protein structure, social networks). As the contents of Panels 2 and 3 are domain-dependent and manually created, the GUI needs to be reconstructed when the domain changes in order to accommodate new domain-specific patterns and labels.

1.2 Our Vision

The common theme that runs through the limitations mentioned above is that *visual query interface construction and maintenance should be dynamic and data-driven rather than pre-selected or manually constructed.* In this paper, we articulate a vision shaped by two fundamental questions.

1. How can we automatically generate and maintain the contents of relevant panels in a visual query interface?
2. Can data-driven visual query interfaces enhance usability and portability across graph repositories?

Specifically, our vision calls for a generic data-driven approach to address the aforementioned limitations associated with the construction and maintenance of traditional data-unaware visual graph query interfaces. A data-driven paradigm has several benefits such as superior support for visual subgraph query construction, significant reduction in the manual cost of maintaining an interface for any graph-based application, and portability of the interface across the diverse variety of graph querying applications. To the best of our knowledge, prior to our recent publication [43], we are not aware of any systematic endeavor of making visual query interface construction and maintenance data-driven.

1.3 Paper Organization

The rest of the paper is organized as follows. In Section 2, we describe the generic structure of a visual graph query interface. Section 3 introduces the architecture of a *data-driven* visual graph query interface construction and maintenance system and key novel research challenges that need to be addressed in order to realize it. Section 4 briefly presents the related research and highlights the novelty of our vision. We briefly report our initial effort to realize this vision in Section 5. The last section concludes this paper.

2. STRUCTURE OF A QUERY GUI

Many visual interfaces for graph query construction [9,19,20,40] are comprised of at least four key panels.

1. An *Attribute Panel* to display the set of labels or attributes of nodes or edges of the underlying data. For simplicity, we assume that this panel consists of a set of node or edge labels[4].
2. A *Pattern Panel* to display the set of canned patterns that can aid query formulation.
3. A *Query Panel* for constructing a graph query graphically by adding a node or canned pattern iteratively.
4. A *Results Panel* that displays the query results.

Figure 2 depicts a screen dump of such a visual interface for querying a set of data graphs (chemical compounds). A typical query would be constructed using the interface by performing the following sequence of steps.

1. Move the mouse cursor to the *Attribute* or *Pattern Panel*.
2. Scan and select a label or pattern (*e.g.,* label C, benzene ring pattern).
3. Drag the selected item to the *Query Panel* and drop it. Each such action represents formulation of a single node or a subgraph in the query graph.
4. Repeat, if necessary, Steps 1–3 for constructing another node or subgraph.
5. Construct edges (if necessary) between relevant nodes in the constructed subgraphs by clicking on them.
6. Repeat Steps 4 and 5 until the complete query graph is formulated.

[4] Attributes associated with nodes or edges of multi-attribute graphs can be easily added in the panel.

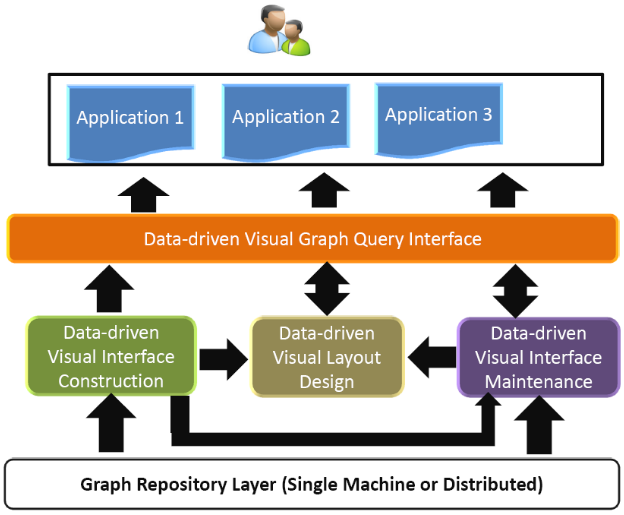

Application 1
Application 2
Application 3
Data-driven Visual Graph Query Interface
Data-driven Visual Interface Construction
Data-driven Visual Layout Design
Data-driven Visual Interface Maintenance
Graph Repository Layer (Single Machine or Distributed)

Figure 3: Framework of data-driven visual graph query interface construction and maintenance.

Observe that the contents of a *Query Panel* and a *Results Panel* are very much dependent on a user and a specific query, respectively. In contrast, the contents of the *Attribute* and *Pattern Panels* depend on the data. *Hence, data-driven construction and maintenance of a visual graph query interface must focus on constructing and maintaining the contents of Attribute and Pattern Panels automatically from the underlying data.*

Additional Panel. Although the aforementioned panels are prevalent in many real-world visual graph query interfaces, some recent interfaces provide an additional *Suggestion Panel* to support intelligent suggestions and feedbacks during graph query formulation. For example, VIIQ [18] automatically suggests top-*k* new edges relevant to a user-specified partial query graph on the *Query Panel* and displays label suggestions for newly added nodes and edges in a *Suggestion Panel*.

3. NOVEL RESEARCH CHALLENGES

Figure 3 depicts the framework of a system for realizing our proposed vision. At the bottom of the figure, graph data is the foundation used to construct and maintain a visual interface, and influences the layout in the interface. At the top of the figure, applications (users) interact with the developed interface. In this section, we discuss in detail the novel research challenges associated with realizing this framework. We first discuss these challenges by assuming that the underlying graph repository contains a large collection of small or medium-sized graphs (Sections 3.1 and 3.2). Then, in Section 3.3 we reconsider these challenges in the context of massive networks (*e.g.,* social network). Note that it is important to distinguish between these two types of graph-structured data that are prevalent in many real-world applications as several recent studies have shown that data management and analytics techniques designed for one cannot be directly adopted to handle the other [15]. Unless specified otherwise, we assume that query logs are not available[5].

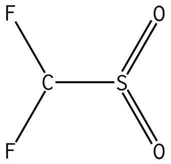


Figure 4: An infrequent canned pattern.

3.1 Data-driven GUI Construction

As remarked earlier, traditionally, data is not used to construct or maintain a visual graph query interface. This limits the usability of an interface and also curtails the portability of the interface since it needs to be manually reconstructed or modified to facilitate visual querying in a new domain. We propose a data-driven strategy to overcome the limitations of the classical approach. Our goal is to *generate the contents of the Attribute and Pattern Panels automatically.* Note that the content of the *Suggestion Panel* is only relevant when a query is being formulated and not during GUI construction.

While the set of labels or attributes of nodes or edges of the data graphs (displayed on the *Attribute Panel*) can be easily generated by traversing them, automatically generating the set of canned patterns (for *Pattern Panel*) is computationally challenging. These patterns should not only maximally cover the underlying graph data but should also minimize topological similarity (redundancy) among the patterns so that a diverse set of canned patterns is available to the end users for query formulation. Note that there can be a prohibitively large number of such patterns. Hence, the size of the pattern set should not be too large due to limited display space on the GUI as well as the inability of users to absorb too many patterns during query formulation.

As some of the canned patterns may be frequent in the underlying data, at first glance it may seem that they can be generated using any frequent subgraph mining algorithm [39]. For example, both GUIs in Figures 1 and 2 contain the benzene ring as one of the canned pattern since many chemical compounds contain the ring. However, this is not the case as it is not necessary for all canned patterns to be frequent. It is indeed possible that some patterns are frequently used by end users to formulate visual queries but are infrequent in the collection. For example, although the pattern in Figure 4 is an infrequent subgraph in *Pubchem*, it can be useful in formulating queries to retrieve different types of (fluorosulfonyl) acetic acid compounds having antiviral properties. Notice that this pattern does not appear in the GUI in Figure 1. Furthermore, a frequent subgraph mining technique may generate a prohibitively large number of patterns, which may be too large for the limited display space of the GUI and also may not necessarily minimize redundancy as well as maximally cover the underlying graph data. Graph summarization techniques [21, 22, 37], which focus on grouping nodes at different resolutions in a large network, cannot be adopted here as these techniques do not focus on generating a concise canned pattern set by maximizing coverage while minimizing redundancy under a GUI constraint. In summary, a novel strategy is necessary to address the problem of data-driven GUI construction.

Given a large collection of data graphs, a data-driven strategy for constructing the contents of the *Pattern Panel* can be as follows. First, we partition the set of graphs into a set of *clusters* where the *topological similarity* among the graphs in a cluster is high, and the similarity is low for graphs in other clusters. Since computing such similarity for all pairs of data graphs can be prohibitively expen-

[5] This assumption is reasonable; otherwise we have to defer realization of our paradigm until the query log is sufficiently large. Furthermore, such log-based techniques are not effective in handling ad hoc query formulation.

sive [35], it is paramount to discover certain topological feature-based *similarity bounds* of the data graph pairs (wherever possible) to allocate them in the correct cluster without computing the similarity scores. However, if the search of effective similarity bounds becomes impractical, then a sampling-based technique may be leveraged to identify a set of *representative* data graphs (which is significantly smaller than the entire collection) and subsequently cluster these representative graphs instead.

Next, the data graphs in each cluster can be *merged* into a single graph called a *closure graph* based on their topological similarities. Intuitively, a closure graph "summarizes" the content of each cluster. The intuition behind this step is that each cluster represents topologically similar data graphs, so it is sufficient to generate a concise and accurate closure graph to represent the cluster in contrast to attempting to finding such closure graphs directly from the underlying repository. Lastly, a collection of canned patterns to be displayed in the GUI can be extracted by traversing these closure graphs. Note that the set of *candidate* canned patterns that maximally cover these closure graphs may be too large to fit in the limited space of the GUI. Hence, it is important to explore effective strategies to *select* a subset of these patterns by minimizing redundancy and maximizing coverage of the patterns for a given *GUI constraint*[6]. The selected patterns also need to ensure superior *GUI aesthetics* (elaborated in Section 3.4).

Our strategy for data-driven construction of canned patterns can also take advantage of in-memory systems [44]. Observe that each cluster can be processed in parallel for closure graph generation as well as canned patterns extraction. Hence, it is interesting to explore how data-level parallelism and shared-memory scale-up parallelism of in-memory systems can be exploited to enhance the performance of the data-driven GUI construction. Specifically, the aforementioned approach needs to be redesigned in the context of the availability of SIMD, bit-parallel algorithms, and on-chip hardware accelerators such as GPUs and FPGAs.

It is worth noting that in practice the construction of a GUI is carried out prior to querying the underlying graph repository. Hence, it is realistic to assume that query logs and usage patterns of users are unavailable to facilitate canned pattern generation.

3.2 Data-driven GUI Maintenance

The preceding subsection focuses on extracting contents of the *Attribute* and *Pattern Panels* from a specific snapshot of the underlying graph data. However, real-world graphs are dynamic in nature. A recent study [45] described approximately 4,000 new structures were added daily to the *SCI finder* database[7]. In the presence of rapidly-growing or changing data, the contents of the *Attribute* and *Pattern* Panels can grow stale quickly. But running the data-driven visual query interface construction technique frequently would be expensive. So it is important to develop an efficient, incremental approach that can dynamically update the contents of these two panels as the underlying database evolves.

There have been several studies on mining evolution of graph-structured data [4]. Yuan *et al.* [41, 42] proposed an incremental technique to update the indexed *graph features* for subgraph search in response to database updates. None of these efforts focus on updating "summary" patterns in an evolving graph repository. Recent efforts on graph summarization [21, 22, 37] do not focus on updating them with changes to the underlying data.

Given a set of existing canned patterns and labels, and a set of updated data graphs, the goal here is to *incrementally* maintain the labels and canned patterns. To this end, a key challenge is to maintain the clusters so that updated closure graphs, labels, and canned patterns can be generated. One possible strategy is to exploit the closure graphs instead of individual data graphs to determine the cluster membership of an updated or new data graph. This is more efficient since the number of closure graphs is typically significantly smaller than the number of data graphs. Specifically, for a new data graph, an existing closure graph that "best" matches the new graph needs to be identified. If there does not exist any such closure graph, then the new graph can be assigned to a new cluster and a new closure graph is computed. Otherwise, it is assigned to the *best matching* cluster and the corresponding closure graph needs to be updated. It is important to explore upper and lower bounds for the matching function to prune matching with irrelevant closure graphs. If an existing data graph is deleted or modified then the corresponding cluster and its closure graph is updated accordingly.

Once the closure graphs are updated, the *Attribute Panel* can be updated in a straightforward manner by scanning the updated closure graphs to identify new labels or remove obsolete labels. The canned patterns can be maintained by traversing these updated closure graphs to seek patterns that can be added to or removed from the *Pattern Panel*. This requires optimization of an objective function that determines whether after addition of a new pattern or removal of an existing pattern, there is gain in coverage and reduction in redundancy in the canned pattern set. Once again it is important to explore the upper and lower bounds of the objective function to filter irrelevant candidate patterns.

Observe that a key challenge to overcome is that the proposed solution must efficiently update the *Attribute* and *Pattern Panels* in real time as such updates may be performed frequently depending on the evolutionary characteristics of the underlying graph repository. Although label updates can be efficiently handled, updating canned patterns can be challenging due to the complexity of the problem especially when the updates are large and occur frequently. Similar to the data-driven GUI construction, one way to address this challenge is to design algorithms that can leverage data-level parallelism and scale-up parallelism of in-memory systems [44]. If updates are so frequent that real-time update is still too expensive, it is possible to batch the updates and perform them periodically at fixed intervals. Note that the challenge here is to identify the "best" periodicity, which is application and workload dependent. Specifically, machine learning-based techniques can be explored to mine the temporal history of updates of the underlying repository to predict the "best" interval for canned pattern maintenance. Since real-world graph data sources may be frequently updated, such information can be used as the ground truth for training the algorithms.

When sufficiently large query logs are available the aforementioned maintenance techniques can be extended to incorporate them. For instance, if a certain pattern rarely appears in past queries then it may be given lower priority to appear in the *Pattern Panel*. Similarly, if certain data graphs are rarely retrieved by user queries, then canned patterns affected by changes to these data graphs can be *lazily* maintained by deferring their maintenance. Two key challenges here are to devise efficient data structures to represent a large collection of graph query logs in a compressed form by identifying common subgraphs and to create a judicious, efficient canned pattern maintenance strategy that *integrates* these query logs and closure graphs to produce a superior set of canned patterns for a given *GUI constraint*.

[6] The GUI constraint can be expressed as follows: the GUI can only display canned patterns having size ranging from k_1 to k_2 (*e.g.,* 2 to 8) and the maximum number of patterns is M for each size.

[7] `www.cas.org/products/scifinder`.

3.3 Data-driven GUIs for Massive Graphs

The aforementioned research challenges are targeted for a single machine-based graph repository containing a large collection of small or medium-sized graphs. As remarked earlier, several real-world graphs (*e.g.,* social networks, Web graph, road networks) do not fall into this category of graphs as they are typically modelled as massive networks containing millions or billions of nodes and edges. Processing such massive graphs in a distributed computing environment (instead of a single-machine system) often yields better performance [23]. Consequently, the single machine-based solutions devised to address the preceding aims cannot be directly used for data-driven construction and maintenance of visual interfaces on such networks.

There have been efforts to build distributed graph computing systems for data processing and analytics applications [1, 23, 26, 28]. These systems are built on top of a shared-nothing architecture and have typically focused on PageRank computation [28], computing connected components [32], subgraph search [13, 27, 36], and data mining [24]. None of these efforts focus on data-driven visual query interface construction or maintenance.

A possible approach to address this problem is to use an appropriate distributed graph computing framework (*e.g.,* Pregel [28], Giraph [1]) to evenly partition the vertex or edge set of the massive network into *M* machines. A keen reader may observe that a single partitioning technique may not always be the "best" choice for all applications. For instance, vertex partitioning may be a good choice for uniform-degree graphs (*e.g.,* road networks) but may lead to computation and communication imbalance for power-law graphs (*e.g.,* social networks) [23]. In fact, for the latter case edge partitioning is a better choice [23]. Hence, given a massive graph for a specific application, we first analyze its topology, and subsequently use the topology to select the partitioning framework.

After partitioning, each machine in our framework contains a large subnetwork of the original massive network. The subnetwork residing on each machine can be further partitioned into a set of subgraphs, which can easily be done in parallel over *M* machines. Observe that a partition on a machine is similar to a data graph representing a small or medium-sized graph. Hence, conceptually each machine now contains a set of small or medium-sized graphs.

Since each machine now contains a set of "data graphs", we can adopt the cluster and label generation strategies discussed in Section 3.1 to generate the clusters and labels on each machine. Note that this step is independent for each machine. Hence, it can be done in parallel. Particularly, three levels of parallelism [44] can be exploited to construct the canned patterns of a GUI.

The labels, on the other hand, are first generated in each machine and then merged using message passing to create a distinct list of labels. Next, the data graphs in each cluster in each machine can be combined into a closure graph by adopting the strategy discussed earlier. Hence, each machine now contains a set of closure graphs. Note that due to the distributed nature of the problem, a closure graph may occur in multiple machines. Consequently, it is important to devise an efficient strategy to merge the closure graph collection in multiple machines by removing identical graphs. This requires exploration of a message passing-based solution that minimizes the number of messages sent over the network yet achieves the desired goal. Lastly, given the *GUI constraint*, the canned patterns needs to be generated in parallel from the closure graphs residing in each machine. Similar to the above step, a canned pattern may be generated in different machines as it may occur in multiple closure graphs residing in different machines. Hence, a message passing-based solution is required so that a pattern is only extracted once.

The strategy for maintaining labels and canned patterns when the data changes also needs to be designed by leveraging the distributed nature of the data. Given a set of machines that contains the modified subnetworks, the set of clusters affected by the updates needs to be identified and the update strategies for corresponding clusters and closure graphs as highlighted in the preceding subsection need to be extended to handle this challenge. Note that, similar to the above step, we need to ensure that the resultant collection of closure graphs in multiple machines is distinct as a modification may lead to the generation of identical closure graphs. The labels and canned patterns can be updated accordingly by leveraging the update strategy in Section 3.2 while ensuring distinct sets of labels and patterns are generated using message passing.

One of the key challenges for the above strategy is that the message passing-based techniques need to effectively remove unnecessary canned patterns or closure graphs without adding significant cost due to messages exchanged. A potentially viable alternative may be to use a master machine for serial computation. Specifically, the closure graphs from all machines are sent to the master, which then aggregates these graphs by removing duplicates and sent them back to the workers. Similarly, the canned patterns or labels from each machine are sent to the master, which then selects the final list of patterns/labels based on the *GUI constraint*.

3.4 Data-driven Visual Layout Design

Task complexity-aware visual layout design. Given the set of labels and canned patterns to be displayed on the GUI, their judicious layout will facilitate fast and easy visual query formulation. It is well-known in the HCI community that how items are laid out impacts the *item selection time* [10, 11]. For instance, in Figure 1 the labels and patterns are laid out in a matrix format whereas in Figure 2 the labels are arranged as a list of items and the patterns are grouped by size and laid out in tabbed panels. The former layout scheme is useful when there are only a small number of labels and patterns while the latter is suitable for displaying larger collections of items. Recall that a limited choice of labels or patterns may adversely impact visual query formulation time and make these panels less useful. Hence, here we seek to answer the following question: *Given a graph repository, what is the "optimal" layout scheme for the labels and patterns that can greatly facilitate fast and easy formulation of visual queries?* Note that the layout design of the current generation of GUIs for graph querying is not data-driven and is based either on arbitrary choices or some simple HCI or aesthetic rules.

Any effective solution to the aforementioned question needs to consider the following two key factors, (a) availability of canned patterns and labels on the GUI; and (b) *search time* to find relevant patterns and labels during query formulation. For example, too few canned patterns would decrease the usability of the GUI in Figure 1. Our data-driven strategies discussed in Sections 3.1 to 3.3 can greatly alleviate this issue. The second factor, *i.e., search time*, is critical, a user who spends too much time searching for patterns or labels during query formulation will be deterred from using the GUI. If a user has extensive knowledge of the GUI, either through prior experience or because the items (patterns or labels) are organized "optimally," then she will be able to find her target item rapidly. On other hand, if the user is unfamiliar with the items (*e.g.,* the items are randomly organized or its organization is unknown), then she has to visually inspect each item in the panels to find the desired item, which increases task completion time. For example, in order to search for a pattern in Figure 2, a novice user needs to click on the size-specific tab and then visually search each pattern in the list to choose the desired one. On the other hand, in Figure 1 she can find the target pattern relatively quickly.

We envisage that a data-driven approach to designing the layout of these items can greatly reduce the search time. Rather than organizing the patterns by their size, it may be more effective to display *separately* patterns that are *most likely* to be used for formulating queries by a wide variety of users. For instance, if the family of benzene rings (*e.g.,* chlorobenzene) frequently appear in queries then it makes sense to have them readily available to the users. Hence, the technical challenge here is to *devise efficient query-log-oblivious strategies to find patterns that are most likely to appear in queries from the collection of canned patterns*. This requires analysis of the closure graphs to identify patterns that are topologically diverse but appear frequently in different locations of the data. In the case of availability of query logs or *user access patterns* (of canned patterns), the strategy can be further refined by mining these data to identify canned patterns that have been accessed together in the past in order to formulate a variety of queries. On the other hand, instead of organizing the labels randomly (*e.g.,* Figure 2), they can be first sorted based on their labels and then based on their frequency of occurrence in the data. This will help a user to move rapidly to a target region and then visually search frequently appearing labels first in the *Attribute Panel*.

Additionally, it is worth exploring techniques to improve the search time by *dynamically adapting* these panels during query formulation. Given a visual fragment of a query Q currently formulated by a user, certain items can be disabled (by making them transparent or gray) in real time if they are not going to be constructed in the subsequent steps in order to return a non-empty result set for Q. For instance, if a user has drawn a `C` node then we can disable the `Pa` item in the *Attribute Panel* as the latter is not in the local neighborhood of `C` in the underlying data. Similarly, certain patterns can be disabled based on their "distance" from the current query fragment matches in the underlying data. Note that since the closure graphs preserve the topological connectivity between nodes and patterns, they can be efficiently leveraged to achieve such dynamically adaptive layout of items. Observe that disabling patterns or labels allows a user to skip these items rapidly and move to a target region, thereby reducing the search time in subsequent steps.

Another alternative strategy towards reducing the query formulation task complexity is to automatically identify top-k canned patterns in the *Pattern Panel* that a user is most likely to use in the subsequent step during query formulation and display them in the *Suggestion Panel* (Recall from Section 2) for easy access. Observe that availability of such suggested patterns eliminates the search time required to seek for these patterns in the *Pattern Panel* during query formulation. As above, given a partially formulated query and the canned pattern set, the closure graphs can be analyzed to provide such suggestions. Furthermore, if query logs or history of *user access patterns* are available, the suggestion generation strategy can further exploit these data to populate the *Suggestion Panel*. For instance, if we know that two canned patterns are frequently accessed together during query formulation by analyzing user access patterns, then whenever a user selects one of them, the other can be given higher priority in the top-k suggestion list in the *Suggestion Panel*. The recent query log-driven effort in [18], which suggests edge increments to the current query graph, is a step in this direction. However, since it provides only edge suggestions, the query formulation process may take many steps and users can only choose limited structural information in comparison to canned patterns.

Aesthetics-aware visual layout design. The aforementioned issues of data-driven visual layout design only aim to reduce the complexity of query formulation task. An issue of equal importance to an end user is the aesthetics of the layout. People prefer attractive interfaces and the effect of aesthetics in GUI appreciation is significant [12]. Existing approaches to make a GUI stand out is to work out all of the details of visual design manually. That is, the layout of a visual query interface is not automatically generated by considering various GUI *aesthetics metrics*. *Hence, how can we make data-driven visual layout design not only task complexity-aware but also visual aesthetics-aware?*

Many HCI studies have asserted a strong link between visual complexity and aesthetics of web pages [38] and have attempted to measure their aesthetics automatically by analyzing HTML sources and screenshots of web pages [33]. In particular, GUI screenshot-based measure is considered superior to other methods as it better represents what a user sees [33]. The work in [29–31] proposed an array of *aesthetics metrics* to quantify visual complexity such as visual clutter, color variability, contour congestion, and layout quality. Hence, the data-driven visual layout design problem can be reformulated as an optimization problem where the goal is to find an "optimal" layout that minimizes query formulation task complexity and visual complexity of the interface. Observe that the visual complexity can be automatically quantified by extending the aforementioned aesthetics metrics, originally designed for web pages, to handle visual graph query interfaces. Furthermore, notice that metrics such as visual clutter and layout quality are influenced by the number of canned patterns and labels on the *Pattern*, *Attribute*, and *Suggestion Panels*. Hence, the data-driven selection and maintenance of the patterns and labels discussed earlier needs to also consider aesthetics metrics so that the visual complexity of the GUI is not adversely impacted.

3.5 Quantitative Models for Performance Study

Lastly, in order to systematically evaluate the effectiveness of our vision of data-driven visual query interface construction over its classical counterpart, it is paramount to undertake an exhaustive empirical study across different datasets and applications. While traditional measures such as efficiency and scalability of various techniques to realize the aforementioned components are naturally important to evaluate the framework, we believe that systematic empirical study to understand user experiences with data-driven visual query interfaces is paramount for the success of the proposed paradigm. At first glance, it may seem that such an investigation can be performed by simply undertaking a user study to measure whether data-driven visual query interfaces provide superior experience in query formulation as well as GUI appreciation compared to classical query interfaces. We believe that subjective studies provide good data for modeling user preferences, but it is also important to incorporate objective analyses for at least two reasons.

- First, it is often prohibitively expensive and time-consuming to engage a large number of users to either formulate a large number of visual subgraph queries on a set of visual interfaces (across different applications) or survey aesthetics of such visual interfaces. This is especially true for small companies, start-ups, small academic research groups, and individual developers and researchers.
- Second, objective methods, such as quantitative models, are less influenced by individual users. That is, they can capture the actual behavior of a user with the visual interface rather than the user perception of it. Note that subjective measures such as user studies are susceptible to a variety of evaluation conditions (*e.g.,* cognitive load) that can impart variations in the outcomes of the study.

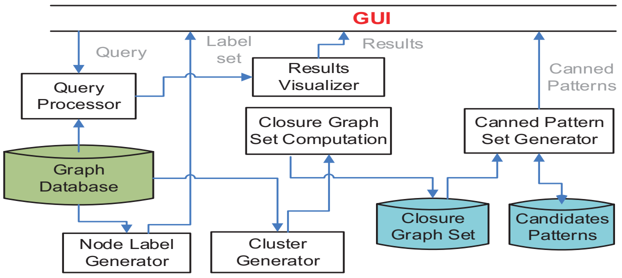


Figure 5: Architecture of DaVinci: our initial effort.

Thus, we emphasize the need for objective measures. In particular, a quantitative model for measuring and comparing the benefits of data-driven visual query interfaces over classical query interfaces (in terms of task complexity and visual aesthetics) needs to be integrated with the traditional subjective usability assessment methods to enable us to systematically investigate the effectiveness of our proposed vision.

Although there is recent work towards building quantitative models for realistic simulation of visual subgraph query formulation [7], to the best of our knowledge, similar effort is missing in the context of visual query interface construction, maintenance, and layout design.

4. RELATED WORK & NOVELTY

Several visual graph querying systems have been proposed to query graph-structured data [9, 18–20, 40]. However, all these approaches follow the conventional paradigm of visual query interface construction and maintenance. In particular, our vision jettisons the longstanding and traditional visual query interface construction paradigm, and takes a data-driven approach to construct and maintain its key components.

In [6], we laid down the vision of *integrating* DB and HCI (referred to as *HCI-aware data management*) techniques towards superior consumption and management of data. Specifically, we presented a "synopsis" of a variety of issues in order to realize HCI-aware data management such as data-driven visual query interface management, visual action-aware indexing and query processing, and HCI-driven visual query performance simulation. In particular, it was skewed more towards the paradigm of making visual query formulation and processing HCI-aware. In contrast, in this paper we focus on challenges and opportunities related to data-driven visual graph query interface construction and maintenance, which is a component of visual query interface management. Specifically, [6] gives a high-level summary of our discussion in Section 3.1 although it did not emphasize on the role of modern hardware in facilitating data-driven construction of different GUI panels. Additionally, except for canned pattern suggestion during query formulation, it did not lay down the vision for strategies related to data-driven GUI maintenance (Section 3.2), strategies for realizing it on massive graphs (Section 3.3), data-driven visual layout design (Section 3.4), and quantitative models for performance study (Section 3.5). In fact, the HCI-driven performance simulation discussed in [6] focuses on simulation of visual query formulation [7] instead of quantitative study of usability and aesthetics of visual query interfaces generated in a data-driven manner. In summary, the vision described here complements the one reported in [6].

The HCI community has made significant progress in studying various issues related to user-friendly visual interface design such as task modelling [34], menu design [5, 10], and pointing and selection activities [3]. However, the HCI community has not taken a data-driven strategy to create these visual interfaces. Algorithmically, as highlighted in Section 3, our vision raises several novel and intriguing research challenges that have not been addressed before.

Lastly, the emphasis of GUI independence for *Model-View-Controller (MVC)* design pattern [8] in the software engineering domain is orthogonal to our vision. The GUI (view) in MVC is a visualization of the model (the data). Generic views can be automatically generated, though users can also program views as desired. When the user needs to program a view, they often use a different kind of GUI (*e.g.,* Ruby-on-rails) to quickly and easily program the view. A visual graph query interface is a GUI in the latter sense (making it easy to program a view of the data, *e.g.,* a Ruby-on-rails for graph databases) and not in the former sense (a constructed view in MVC). Hence, the data-driven approach to building a visual graph query interface is akin to automatically tailoring the Ruby-on-rails GUI to a specific data collection, an application constructed using this tailored GUI would still implement an MVC approach.

5. INITIAL EFFORT

In the last two years, we have investigated this novel paradigm and made the first effort to realize its feasibility by building the data-driven visual graph query interface construction component called DaVinci [43][8] for a large collection of small or medium-sized graphs. It is implemented in Java and realizes some of the strategies discussed in Section 3.1. Specifically, it automatically generates the contents of the *Attribute* and *Pattern Panels* of a visual graph query interface from the underlying database.

Figure 3 shows the architecture of DaVinci[9]. The *Node Label Generator* module traverses the underlying collection of data graphs $\mathcal{D}$ (*e.g.,* chemical compounds) to generate the set of unique labels in it, which are then displayed on the GUI. The *Cluster Generator* module constructs clusters of data graphs from $\mathcal{D}$ where the similarity among data graphs in the same cluster is high while it is low for graphs in different clusters. We use *maximum connected common subgraphs* (mccs) [35] to compute similarity between a pair of graphs. First, a pair of data graphs g_1 and g_2 are randomly chosen from $\mathcal{D}$ that have potentially least similarity by utilizing data graph features (for reason discussed below). This pair is used as two *pivots* for clustering. For all remaining data graphs $g_i \in \mathcal{D}$, we sort them in ascending order based on the difference of their similarity scores w.r.t g_1 and g_2 and associate the first half of the sorted list to g_1 and the rest to g_2. This strategy is recursively carried out until the size of the cluster is below a specific threshold. As remarked earlier, computing similarity scores of all pairs of data graphs can be prohibitively expensive. Hence, we utilize some simple heuristics (*e.g.,* size of data graphs, label set similarity) of the data graph pairs to allocate a data graph in the correct cluster whenever possible, without computing the similarity scores. For example, if the size of data graph is significantly different from g_1 then it is highly likely that they are dissimilar and hence is put in the cluster of g_2, without computing the similarity scores. Similarly, if the label set of a data graph is significantly different from g_1 then it is directly put into the other cluster. For the remnant data graphs we sort them based on the similarity scores.

[8] DaVinci is demonstrated in IEEE ICDE 2015 [43].

[9] The *Query Processor* and the *Results Visualizer* modules are used to evaluate the formulated query and display the query results, respectively. Hence, they are orthogonal to our vision and were only included in DaVinci to "close-the-loop" for demonstration purpose.

The *Closure Graph Set Computation* module combines all the data graphs in each cluster into a single graph called the *closure graph* that "summarizes" the content of each cluster as highlighted in Section 3.1. Currently, we extend the idea of *graph closure* in [16] to compute it. First, for each cluster we create a *mapping* between a pair of data graphs (g_1, g_2) by *extending* each data graph with dummy vertices and edges such that each vertex and edge in g_1 has a corresponding *mapping* in g_2 based on label and edge matching. A dummy vertex or edge is assigned the label ε and each non-dummy vertex and edge is annotated with the identifier of the original data graph it belongs to. Next, given two such extended graphs and a mapping between them, its closure graph $g_c(V_c, E_c)$ is constructed where the attribute of a vertex (*resp.* edge) in g_c is union of the attributes of the corresponding mapped vertices (*resp.* edges) of the extended graphs. All the matchings between the vertices and edges are established by computing the similarity between each pair of vertices using the *Neighbor Biased Mapping (NBM)* [16], which bias the matching towards neighbors of already matched vertices. Each vertex (*resp.* edge) in g_c is also annotated with the union of the data graph identifiers of the corresponding mapped vertex (*resp.* edge) pairs. The dummy labels are removed from the closure graph. The final closure graph to represent the set of data graphs in a cluster is built recursively from the data graphs and the closure graphs.

Lastly, the *Canned Pattern Generator* module extracts a collection of canned patterns from the set of closure graphs, which are displayed on the GUI grouped by their size. It consists of two key steps, namely, *candidate pattern set generation* and *canned pattern set selection*. In the first step, we find candidate patterns (subgraphs) in a closure graph that maximize the objective function $|g|Cov(g)$ where $|g| = |E|$ is the size of a subgraph g and $Cov(g)$ is the *coverage* of g measured as the number of data graphs that contain g. Next, the candidate pattern sets from all closure graphs are aggregated by removing duplicate patterns and aggregating their coverage. Since this candidate pattern set can be too large to fit in a given GUI in its entirety, in the next step a subset of these patterns are selected greedily by maximizing an objective function that maximizes coverage and minimizes similarities among the patterns. First, the candidate patterns are grouped by their size and within each group the pattern p with the maximum coverage is selected. The coverage of each remaining candidate pattern in the group is updated by penalizing it by its similarity to p. This process is repeated until the selected pattern set satisfies the *GUI constraint* (recall from Section 3.1). Finally, these canned patterns are displayed on the GUI (grouped by size).

The road ahead. Our aforementioned initial effort demonstrates the feasibility of the promise of data-driven visual query interface construction. Note that in this effort the canned patterns are generated offline as the cluster generation step is time-consuming. Although we have used simple feature-based heuristics to reduce the computational cost, it still takes a significant amount of time (*i.e.,* several hours) to generate clusters especially for large collection of data graphs. Hence, a more efficient and scalable solution to this problem is an open research challenge that needs to be addressed. To this end, we have not yet explored the alternative sampling-based technique to identify a set of representative data graphs for clustering as remarked in Section 3.1. Furthermore, the current implementation of DaVinci does not leverage on data-level parallelism and shared-memory scale-up parallelism to improve the efficiency and scalability of data-driven GUI construction. Also, our canned pattern selection technique is GUI aesthetics-unaware.

In addition, DaVinci does not currently focus on data-driven GUI maintenance (Section 3.2) and data-driven visual layout design (Section 3.4). Also, observe that our initial effort was built on top of a graph repository consisting of a large set of small or medium-sized data graphs. Hence, realizing these issues on massive graphs (Section 3.3) is an open problem. Lastly, although we received much positive feedback from the audience on DaVinci during our demonstration in ICDE 2015, comprehensive qualitative and quantitative models for performance study (Section 3.5) have not yet been explored. This is crucial for wider acceptance of our vision of data-driven visual query interface construction and maintenance.

6. CONCLUSIONS

This paper contributes a vision of data-driven visual graph query interface construction and maintenance by presenting a visual graph querying framework where the contents of several panels of the GUI are automatically generated and maintained from the underlying data. Specifically, our vision attempts to carve out a substantially new research topic that blends two orthogonal fields, namely data management (graph query formulation) and HCI (visual interface design), to make query interface design data-driven. To the best of our knowledge, this paradigm has not been systematically investigated before, prior to our recent publication.

Measures of success. Successful realisation of this paradigm will enhance the portability and maintainability of visual query interfaces and reduce construction cost. But several non-trivial and novel research challenges need to be overcome to realize the paradigm. To this end, it is paramount to conduct extensive user studies to investigate the benefits of a data-driven approach in comparison to the classical approach of constructing and maintaining visual query interfaces. As discussed in Section 3.5, such a user study should complement quantitative performance models. Furthermore, wide-spread adoption of data-driven query interfaces in lieu of classical interfaces will be another measure of success.

Wider applicability. We focused on graph querying as graphs are a natural way of modeling data in a wide variety of domains. However, it is easy to see that our vision can be adopted for a variety of complex database management systems such as JSON [25] and XML databases, etc. as all these databases use formal query languages that are widely acknowledged for their syntactic complexity [2, 14]. Consequently, the novel research challenges discussed in this paper to realize data-driven visual graph query interface are also relevant to building data-driven visual query interfaces for tree-structured data.

7. ACKNOWLEDGMENTS

Sourav S Bhowmick was supported by the Singapore-MOE AcRF Tier-2 Grant MOE2015-T2-1-040. Byron Choi was partially supported by HKRGC GRF, HKBU12201315.

8. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] S. Abiteboul, et al. The Lowell Database Research Self-Assessment. *In Communication of the ACM*, 2005.

[3] J, Accot, S. Zhai. Beyond Fitts' Law: Models for Trajectory-Based HCI Tasks. *In ACM SIGCHI*, 1997.

[4] C C. Aggarwal, K. Subbian. Evolutionary Network Analysis: A Survey. *ACM Comput. Surv. (CSUR)*, 47(1):10, 2014.

[5] D. Ahlstrom, R. Alexandrowicz, M. Hitz. Improving Menu Interaction: A Comparison of Standard, Force Enhanced and Jumping Menus. *In SIGCHI*, 2006.

[6] S. S. Bhowmick. DB ⋈ HCI: Towards Bridging the Chasm Between Graph Data Management and HCI. *In DEXA*, 2014.

[7] S. S. Bhowmick, H.-E. Chua, B. Thian, B. Choi. ViSual: An HCI-inspired Simulator of Blending Visual Subgraph Query Construction and Processing. *In ICDE*, 2015.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture. Wiley, 1996.

[9] D. H. Chau , C. Faloutsos, H. Tong, J. I. Hong, B. Gallagher, T. Eliassi-Rad. GRAPHITE: A Visual Query System for Large Graphs. *ICDM Workshop* , 2008.

[10] A. Cockburn, C. Gutwin, S. Greenberg. A Predictive Model of Menu Performance. *In ACM SIGCHI*, 2007.

[11] A. Cockburn, C. Gutwin. A Predictive Model of Human Performance with Scrolling and Hierarchical Lists. *Human-Computer Interaction* 24(3): 273-314 (2009).

[12] A. De Angeli, A. Sutcliffe, J. Hartmann. Interaction, Usability and Aesthetics: What Influences Users' Preferences? *In Proc. of Conference on Designing Interactive Systems*, 2006.

[13] W. Fan, X. Wang, Y. Wu, D. Deng. Distributed Graph Simulation: Impossibility and Possibility. *In PVLDB*, 7(12), 2014.

[14] J. Gray, D. T. Liu, M. Nieto-Santisteban, et al. Scientific Data Management in the Coming Decade. *In SIGMOD Record*, 34(4), 2005.

[15] W.-S Han, J. Lee, M.-D. Pham, J. X. Yu. iGraph: A Framework for Comparisons of Disk Based Graph Indexing Techniques. *In VLDB*, 2010.

[16] H. He, A. K. Singh. Closure-tree: An Index Structure for Graph Queries. In *ICDE*, 2006.

[17] H. He, A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. *In SIGMOD*, 2008.

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

[19] C. Jin, S. S. Bhowmick, X. Xiao, J. Cheng, B. Choi. GBLENDER: Visual Subgraph Query Formulation Meets Query Processing. *In SIGMOD*, 2011.

[20] C. Jin, S. S. Bhowmick, B. Choi, S. Zhou. PRAGUE: A Practical Framework for Blending Visual Subgraph Query Formulation and Query Processing. *In ICDE*, 2012.

[21] D. Koutra, U. Kang, J. Vreeken, C. Faloutsos. VOG: Summarizing and Understanding Large Graphs. *In SDM*, 2014.

[22] K. LeFevre, E. Terzi. GraSS: Graph Structure Summarization. *In SDM*, 2010.

[23] A. Lenharth, D. Nguyen, K. Pingali. Parallel Graph Analytics. *Communications of the ACM*, 59(5), 2016.

[24] W. Lin, X. Xiao, G. Ghinita. Large-scale Frequent Subgraph Mining in MapReduce. *In ICDE*, 2014.

[25] Z. H. Liu, B. C. Hammerschmidt, D. McMahon. JSON Data Management: Supporting Schema-less Development in RDBMS. *In SIGMOD*, 2014.

[26] Y. Low, J. Gonzalez, A. Kyrola, et al. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *In PVLDB*, 5(8), 2012.

[27] S. Ma, Y. Cao, J. Huai, T. Wo. Distributed Graph Pattern Matching. *In WWW*, 2012.

[28] G. Malewicz, M. H. Austern, A. J. C. Bik, et al. Pregel: A System for Large-scale Graph Processing. *In SIGMOD*, 2010.

[29] E. Michailidou, S. Harper, S. Bechhofer. Visual Complexity and Aesthetic Perception of Web Pages. *In Proc. of ACM International Conference on Design of Communication*, 2008.

[30] A. Miniukovich, A. De Angeli. Quantification of Interface Visual Complexity. *In Working Conference on Advanced Visual Interfaces*, 2014.

[31] A. Miniukovich, A. De Angeli. Computation of Interface Aesthetics. *In SIGCHI*, 2015.

[32] V. Rastogi, A. Machanavajjhala, L. Chitnis, A. D. Sarma. Finding connected components in MapReduce in logarithmic rounds. *In ICDE*, 2013.

[33] K. Reinecke, T. Yeh, et al. Predicting Users' First Impressions of Website Aesthetics with a Quantification of Perceived Visual Complexity and Colorfulness. *In SIGCHI*, 2013.

[34] D. D. Salvucci, N. Taatgen, J. P. Borst. Toward a Unified Theory of the Multitasking Continuum: From Concurrent Performance to Task Switching, Interruption, and Resumption. *In SIGCHI*, 2009.

[35] H. Shang, X. Lin, Y. Zhang, J. X. Yu, W. Wang. Connected Substructure Similarity Search. *In SIGMOD*, 2010.

[36] Z. Sun, H. Wang, H. Wang, B. Shao, J. Li. Efficient Subgraph Matching on Billion Node Graphs. *In PVLDB*, 5(9), 2012.

[37] Y. Tian, R. A. Hankins, J. M. Patel. Efficient Aggregation for Graph Summarization. *In SIGMOD*, 2008.

[38] A. N. Tuch, E. E. Presslaber, et al. The Role of Visual Complexity and Prototypicality Regarding First Impression of Websites: Working Towards Understanding Aesthetic Judgements. *International J. of Human-Computer Studies*, 70, 2012.

[39] X. Yan, J. Han. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.

[40] S. Yang, Y. Xie, Y. Wu, et al. SLQ: A User-friendly Graph Querying System. *In SIGMOD*, 2014.

[41] D. Yuan, P. Mitra, H. Yu, C. L. Giles. Iterative Graph Feature Mining for Graph Indexing. *In IEEE ICDE*, 2012.

[42] D. Yuan, P. Mitra, H. Yu, C. L. Giles. Updating Graph Indices with a One-Pass Algorithm. *In SIGMOD*, 2015.

[43] J. Zhang, S. S. Bhowmick, H. H. Nguyen, B. Choi, F. Zhu. DAVINCI: Data-driven Visual Interface Construction for Subgraph Search in Graph Databases. *In IEEE ICDE*, 2015.

[44] H. Zhang, G. Chen, B. C. Ooi, et al. In-memory Big Data Management and Processing: A Survey. *In TKDE*, 27(7):1920–1947, 2015.

[45] L. Zou, L. Chen, J. Xu Yu, Y. Lu. A Novel Spectral Coding in a Large Graph Database. *In EDBT*, 2008.