Complex Analytical Queries over Large Attributed Graph Data

Dawit Yimam Seid, Sharad Mehrotra
University of California, Irvine
Irvine, CA, USA
dseid,sharad@ics.uci.edu

Abstract

A number of new analysis and data mining tasks over large and dynamic attributed graph data are becoming popular in a wide variety of applications including web data mining, bio-informatics, intelligence link analysis, and social network analysis. The data manipulation required in these applications is characterized by two features (1) complex analytical queries, (2) queries over both link structures and attribute values. The data models and basic query operators developed by previous work are inadequate in both of these respects. As a result, presently, graph data analysis applications are being implemented inefficiently as memory-based systems that use database systems just for storage of data and not for querying them. In this paper, we address this problem by presenting a complete attributed graph query language equipped with grouping and aggregation operations. Based on this language we define OLAP-like operators that enable hierarchical view based, multi-granular analysis of attributed graph data. We demonstrate the power and generality offered by the composable and reorderable operators of our language through a number of realistic examples. Finally, we give implementations of our operators in relational database systems clearing the way for their eventual incorporation into database systems.

1 Introduction

Numerous analysis and data mining tasks (in a wide variety of applications including intelligence link analysis, social network analysis, inter-organizational link analysis, web data mining, bio-informatics) are based as much on the links among heterogeneous entities and events as the properties of individual entities. In such applications, data is represented and manipulated as attributed graphs where nodes and links represent various types of entities and their relationships, respectively. It is also important in these applications to be able to associate arbitrary number of attributes to the nodes and links and be able to express complex structure-constrained queries. All these requirements can not be met by existing database systems alone, since these systems are purely record based, with relationships between records being a matter for application programs.

Although earlier work in the database field has looked into data models and simple query operators to support graph data querying in database systems, this body of work is not sufficient to support/express the complex graph querying and analysis that is required by a number of novel applications. As a result, presently, graph data analysis applications are being implemented inefficiently as memory-based systems that use database systems just for storage of data and not for querying them. In addition to being cumbersome and error-prone, this approach deprives these applications the numerous efficient query execution support of databases.

Therefore, there is a need to enhance the capability of the database engine to manipulate graph data and associated attribute values for both expressiveness and efficiency reasons. In terms of expressiveness, there is a need for a language over attributed graph data that (1) provides an intuitive way to manipulate relationships/links as first class elements, and (2) allows specification and optimization of complex queries about relationships in a declarative manner in place of the current approach of expressing such queries operationally. Efficiency wise, there is a need to perform computations over graph structure and associated information as close to the data as possible in order to avoid massive data uploads. For example, database-based operators can help in scaling up graph analysis techniques by efficiently executing their matching and sufficient statistics gathering components which commonly take up most of the processing time.

To meet these requirements, we introduce a minimal, yet powerful set of query operators as part of a graph query language called $g$Analytic. We also present an algebra of this
language that permits the composition of query operators to express complex analytical queries over graph databases. In order to address the particular challenges presented by analytical queries, our language also provides powerful grouping and aggregation operations for attributed graphs and clearly defines their semantics. On top of these basic operators, we also define OLAP-like roll-up and drill-down operations over hierarchic views of graphs. Through numerous examples, we show that, due to their composability and reordering feature, the set of operators we define offer tremendous expressive power and a wide range of opportunities for query optimization.

1.1 Motivational Queries

To illustrate that kinds of analytical queries that can be asked over attributed graphs, consider the example graph data shown in figure 1 from an application that aims to analyze citation and collaboration relationships among researchers. This graph consists of heterogeneous node and link types whose schema is shown in the diagram next to the graph data. Consider the following analytical query on this graph: for each institute node, compute a measure of its influence based on the influence (as measured by citations received) of the papers published by its member researchers; use the Hubbel’s metric [15], given by $s_i = e_i + \sum_{j=1}^{m} r_{ij} s_j$, where $s_i$ and $s_j$ are the status or influence of papers $i$ and $j$, respectively, $r_{ij}$ is some measure of strength to which $j$ is influenced by $i$, and $e_i$ is an exogenous value. Notice that the influence measure is to be computed on the citation graph of papers and that measure is somehow aggregated for institutes to find the required measures. A more complicated query can be asked that requires the influence values computed for a given topic or conference.

The challenges presented by these queries are three-fold. First, we need a way to algebraically express the computation of aggregations on the structural property of graphs as is the case with computing the Hubbel’s metric. Second, we need to be also able to express predicates over node or link attributes along with predicates over link structures (e.g. topic of a paper). Third, we need to be able to overlay a hierarchic view on top of the graph data in order to aggregate some metrics which, in turn, can themselves be aggregates at a different level (e.g. aggregating Hubbel’s score of papers to institutes).

2 Data Model

The most attractive property of the attributed graph data model is its equal treatment of link structure and attribute data associated with nodes and links. Basically, an attributed graph [16, 11] is a graph where nodes and links have associated attributes taken from some attribute sets like natural numbers, real numbers or strings. By a graph we mean an undirected unlabelled graph $\mathcal{H} = \langle V, \mathcal{L}, f_{\text{incidence}}, f'_{\text{incidence}} \rangle$ with a set of vertices $V$, a set of links $\mathcal{L}$, and functions $f_{\text{incidence}} : \mathcal{L} \rightarrow V$ and its inverse $f'_{\text{incidence}}$ which associate to each link its incident pair of vertices. We use the term graph element to refer to either a vertex or a link. For a vertex, we use the term adjacent to refer to all nodes that have a link to it in a given attributed graph.

Definition 1 (Attributed Graph.) Let $\mathcal{D}_V$ and $\mathcal{D}_L$ be the global domains of possible values for attributed vertices and links, respectively. An attributed graph $\mathcal{G}$ over $(\mathcal{D}_V, \mathcal{D}_L)$ with an underlying graph structure $\mathcal{H} = \langle V, \mathcal{L}, f_{\text{incidence}}, f'_{\text{incidence}} \rangle$ is a pair $\langle G_V, G_L, f_{\text{incidence}}, f'_{\text{incidence}} \rangle$ where $G_V = \langle V, \psi_V \rangle$ is an attributed vertex set and $G_L = \langle \mathcal{L}, \psi_L \rangle$ is an attributed link set. The mappings $\psi_V : V \rightarrow G_V$ and $\psi_L : \mathcal{L} \rightarrow G_L$ assign attribute values to vertices and links, respectively.

A very useful property of attributed graphs is the flexibility they provide to represent and manipulate graphs composed of heterogeneous nodes and links.

An attributed graph database can be viewed at two levels: at the type (schema) level and at data (instance) level. Below we describe each in detail.

2.1 Schemas

An attributed graph schema $\mathcal{S}$ is defined as a weakly connected labeled multigraph (i.e. a pair of nodes can be connected by more than one edge). Each node and link in the schema represents a graph element type. Each type consists of a set of attributes (i.e. atomic types) which describe the properties of the graph element type. Whereas a vertex type specifies the set of attributes associated to the vertex, an edge type additionally specifies the types of incident nodes.
A "schema path" is a series of links that are followed to get from one node/type to the other in the schema. There are two kinds of links between nodes in a schema: recursive (mono-type) links and non-recursive (bi-type) links.

**Directionality.** As discussed above, for each function that associates a vertex to a link, there is also a reverse function that associates the second vertex to the same link. This definition allows to readily represent directionality in the graph. This approach is motivated by the observation that link directionality is a semantic concept and there is always a valid reverse semantics for every directed link (e.g. Author.authored.Paper, and Paper.authoredBy.Author). To capture both semantics, we require that, for every link, an inverse of it is also should be somehow (e.g. functionally) defined as part of the schema. As will be clear later, this provides flexibility in the expression of patterns over the schema using sets of path expressions.

**Cardinality.** Sometimes it may be useful to associate with a link type constraints over the cardinality of its incident nodes. Such constraints (e.g. one-to-one, one-to-many, etc) can be specified at the schema level and later enforced by the database system. Notice that this is similar to the cardinality constraints in traditional Entity-Relationship (ER) data modeling.

### 2.2 Attributed Graph Database

The Attributed graph database $\mathcal{G}$ is an unlabeled multigraph where each data node (and link) is an instance of a node (link) in the schema. Unlike the schema graph, an attributed graph database can be disconnected and may contain isolated nodes (i.e. nodes that do not participate in any link). Hence, the database is essentially a collection of attributed nodes and links of various types.

There exists a function $\mathcal{F}_S$ from the database $\mathcal{G}$ to its schema $S$ associating with each node (and each link) in $\mathcal{G}$ a nod (a link) in $S$. Conversely, there is an instantiation function $\mathcal{F}_T$ that associates with each node (link) in $S$ a set of nodes (edges) in $\mathcal{G}$.

### 3 Query Model and Processing Flow

#### 3.1 Operands

In any query algebra, it is necessary to define the operands on which operations are to be applied. In relational algebra, these operators are relations composed of tuples. The operands for our algebra are sets of connected attributed graphs (i.e. sets of attributed nodes and attributed edges). Since a path is also a graph, our algebra can also be applied on path sets. Since a single node is considered as a trivial graph, our algebra can also be applied on sets of nodes. If we represent a node as a tuple composed of its attribute values, then our algebra can be applied on tuple sets (relations). Therefore, the graph algebra defined in this paper is a superset of the relational algebra.

#### 3.2 Patterns over schemas

A syntactic mechanism to specify attributes of interest is necessary in any query algebra. This mechanism is needed for two reasons: (1) to specify part of the database on which a particular query is made on, (2) to specify the output from a query. In relational algebra, specifying both of these is straightforward. For queries over attributed graph databases, a corresponding mechanism should somehow enable us to specify subgraphs (patterns) of interest based on a schema. For this purpose, gAnalytic uses regular path expressions (without alterations) on the alphabet of node and link types (labels) in the graph schema since it offers the flexibility required to specify a large class of structural patterns. We refer to this expressions as subschema expressions or pattern expressions. A subschema expression essentially represents a structural projection of the graph schema.

A path expression is a regular expression over $T$ without alternation ($+$), whose language contains only alternating sequence of node and link types which (1) each link is incident with the two nodes surrounding it, and (2) begins and ends with nodes. A path expression $e$ specifies all paths in the attributed graph database whose structure matches $e$. An expression on this path involves the standard operation namely . (dot) as a concatenation operator, Kleene closure ($\ast$) or $e+$ (one or more repeats of expression $e$) which is equivalent to $e.e\ast$, and addition $+$. We also define two new operators: $\text{Any}N$ and $\text{Any}L$ to represent any node or link type, respectively, in the expression.

At this point, one might wonder we define path expressions over both nodes and links instead of, for example, using regular expression only on link labels (as is the case for semi-structured data, XML, etc.). We will discuss this when we describe renaming in section 3.3 but briefly the reason

---

1. Notice that this definition of inverse in our schema definition is similar to the inverse class definition in the OQL [12] language for object-oriented databases.
2. For example, this enables us to expresses paths in any direction thereby avoiding the use of multiple joining paths. Joining paths will be necessary only to express a subgraph composed of truly branching paths.
3. Notice that the operand may also be one large attributed graph (or the entire graph database) on which some complex set of operations can be applied.
4. The language of a regular expression over $T$ is defined as: $L(a) = a$ for all $a \in T$, $L(AB) = L(A) \times L(B) = a, b | a \in L(A) \text{ and } b \in L(B)$ for all expressions $A$ and $B$ over $T$, $L(A + B) = L(A) \cup L(B) = a | a \in L(A) \text{ or } b \in L(B)$ for all expressions over $T$, $L(A\ast) = \bigcup_{i \geq 0} L(A^i)$ where $A^0 = \epsilon$ and $A^i = AA^{i-1}$ for all expressions $A$ over $T$. 

---
has to do with easy expression of various structural constraints in general (for e.g. to specify particular nodes on which intersection of paths occurs) and the ability to distinguish multiple occurrences of same type node in a path expression due to recursive links and express constraints and predicates on a particular node.

Although path expressions are powerful, they alone are, however, not sufficient to express all interesting patterns over the schema. Specifically, it is not possible to express a subschema composed of branching and/or intersecting paths. For this we use the addition operator (+) over path expressions. The semantics of addition here is to return a sum of two path expressions by combining their shared node(s) and return empty set otherwise. Notice that although generally the addition operation can be performed on two paths that have no shared nodes (or paths), in our definition we require that there exist a shared node; the subschema expression is considered illegal otherwise. This restriction simplifies the algebra and allows us to get connected attributed graphs which can be considered as identifiable units (i.e. “tuples”).

Hence, the attributed graphs that constitute the operands of our query operators are expressed using subschemas which in turn are expressed as additions of path expressions. This corresponds to the way a tuple in relational algebra is specified by a set of its attribute names (or the relation name).

### 3.3 Node Renaming in Subschema Expressions

A subschema expression may contain multiple occurrences of same node type (e.g. over a recursive link) and we may need to distinguish them for various reasons. The need for distinguishing multiple occurrences is a fundamental reason for our approach of including both nodes and links in a path expression in the first place (unlike, for example, only a path consisting of links as in semi-structured data and XML).

In particular, The node-link path approach allows:

- syntactic reason: ability to express branching paths flexibly as a sum of two paths. Renaming makes it possible to specify exactly how multiple paths are combined (intersected) to express a subgraph. This is because renaming enables to distinguish points of intersection of multiple paths, which is difficult to express.

- Relabeling of nodes with the same type can enables the expression of various constraints over node labels in the presence of multiple occurrence of the same node type (due to one or more hops of recursion) in a subschema expression. Different node labels can be used for the purpose of easy expression of constraints on node distinctness, e.g. paper.cite.paper, paper1.cite.paper2, paper1.cite.paper2.cite.paper3, etc. Basically, there are three possible constraints: (1) incident nodes must be different, (2) incident nodes must be the same, (3) accept both. Constraint 1 can be expressed using renaming. Constraint 3 is expressed by using same labels (default labels) for both nodes. Constraint 2 translates to matching only cyclic links in the database (i.e. link instances over a single node instance like self-citation link over a paper). However, same constraint may also specified to create longer cycles like a.cite.p.WrittenBy.a. One approach to express constraint 2 is to use the default labels (as in constraint 3) and express the constraint in a selection predicate that requires the ids of the two authors be the same. Notice that constraint 1 also can be expressed in this manner. But renaming is a more elegant approach for this constraint.

- ability to specify predicates on a particular node type despite the presence of multiple occurrences of the same type.

Notice that, unlike attribute renaming in relational algebra, renaming of nodes has the added effect of selecting a subset of the subgraphs that are instantiation of a give subschema.

### 3.4 Query Processing Flow

Here we first present a brief description of the overall approach for queries over attributed graphs. A query is an expression composed of a sequence of expression of the form $O(S)$ or $O(S_1, S_2)$ where $O$ is a query operator and $S_1, S_2$ are subschema expressions. These queries are formulated based on the schema. Before algebraic operators are applied, the subschema operators should be first validated over the schema and then instantiated over the database in order to get the set of subgraphs which are input to the operator. In section 4.1, we will define two operators $SS_{inst}$ and $SG_{inst}$ that perform subschema validation and instantiation respectively.

Once a set of graph(s) of interest is instantiated, it is evaluated by an algebraic operator and a set of graphs is returned as a result. Hence, all operators in our algebra are closed under the set of graphs.

Figure 2 depicts this simple query processing flow. Notice that this is an non-optimized version. In reality, a variety of optimizations can result in the overlap of subschema instantiation and algebraic operations. For example, in a

---

5As we will show later, the addition is implemented using the graph join operator over the graphs which are the instantiations of the involved path expressions.
simple optimization, predicates in a selection operation can be pushed down to the subschema instantiation step.

4 The Operators

In this section we introduce the operators that constitute our attributed graph query language called gAnalytic. Although the main focus of this paper is on the definition of grouping and aggregation operators over the attributed graph model and developing OLAP-like operators on top of them, we will present the entire language here for the sake of completeness. Over the attributed graph model, gAnalytic language defines basic operations corresponding to relational algebra (like selection, projection, join), standard set operations (like union, intersection, difference and Cartesian product) as well as grouping and aggregation operators. The basic operators and set-based operators are inspired by the extensive literature giving similar definitions.

in the subsequent sections, we discuss the operators in three groups: schema operations in section 4.1, basic operations in section 4.2 and grouping/aggregation operations in section 4.3

4.1 Graph Schema Operators

Subschema Instantiation ($SS_{\text{inst}}$). The $SS_{\text{inst}}$ operator takes in a subschema (pattern) expression and generates its instantiation over an attributed graph database schema. It matches each path expression in a pattern and performs a path addition over these paths to make sure that the given subschema is valid over the schema. It returns the data level representations of each node and edge in the expression from which the subschema instantiation is derived.

Subgraph Instantiation ($SG_{\text{inst}}$). The $SG_{\text{inst}}$ operator generates instantiations of the subschema in the attributed graph database. Given a pattern and a set of representations of nodes and edges (results from $SS_{\text{inst}}$), it returns the set of matching instantiations.

In case of a subschema consisting of more than one path expressions, we need a mechanism to perform the subschema composition and instantiate the proper subgraphs. As it turns out, the join operator to be defined later in section 4.2.3 can achieve exactly this. In other words, the addition operation at the path expression level translates to a graph join operation at the attributed graph database level.

For a path expression that contains a closure symbol, $SG_{\text{inst}}$ computes a transitive closure $^6$. Notice that for a closure to have a meaning, the two extreme nodes must be of the same type. This is automatically true for a single link. But, it also applies for longer paths.

4.2 Basic Query Operators

4.2.1 Selection

A selection operator, denoted as $\sigma_C(S)$, evaluates a Boolean condition $C$ on subschema labels. On the attribute set of each node/edge in a subschema, conditions can be specified exactly like relational databases (i.e. involving combination operators like $\land$, $\lor$, and $\neg$).

Notice that when we specify a subschema, we are essentially performing a selection of a subset (subgraph) from the attributed graph. However, this selection was purely based on structural properties of the attributed graph. The selection operator being introduced here, instead, uses predicates on the attribute data part of attributed graphs. Hence, at least algebraically, we apply structural filtering first followed by attribute values based filtering; though this order can be changed by a query optimizer.

Selection predicates can be divided into two based on the number of nodes/links they are defined on.

1. predicates on attributes of a single node or edge.

2. predicates defined on multiple nodes, edges or both. (e.g. Find all citation links where cited and "citer" are published same year; find all citation links where the number of citations received by the 'citer' is less than that of the cited). Given two graph element labels $t$ and $t'$ of the same type present in the subschema expression, the predicate $t = t'$ or $t \neq t'$ is defined as a predicate over the graph element's key (id) attribute. By using this types of predicates, one can express a subset of links that, for example, are cyclic (like self-citation over a paper type can be expressed by a predicate that states the ids of the two nodes in

$^6$We will see later that aggregation predicates on the length of paths with a closure can limit the depth of the transitive closure. The query optimizer will have to push those conditions down to the subgraph instantiation.
a cite link be the same). As stated in section 3.2 the constraints expressed using renaming can also be partially expressed using such predicates (except specifying points of intersection of two paths containing multiple occurrences of the same type).

A selection predicate over a node or an edge in a closure sub-expression is applied on all instances of it.

### 4.2.2 Projection

Given a sub-expression, \( E \) of a given subschema expression \( S \), projection, \( \pi_E(S) \) extracts instantiations of \( E \) eliminating the remaining nodes and links in \( S \). In order to simplify the semantics (and implementation), we two restriction are imposed on the structure of \( E \) [1]: (1) \( E \) should be a legal subschema (i.e. be connected), (2) if \( S \) contains multiple occurrences of the same type which is also shared by two path expressions in the subschema, then all of its occurrences must be included in \( E \). The second condition is a mechanism to achieve connectedness (i.e. condition 1) when multiple path expressions are used.

**Example 1** Over the schema given in figure 1, the query in figure 3 finds those researchers that have a faculty position in an institute located in California.

### 4.2.3 Product and Join

In order to define the product and join operators, we need to first develop a method to compute the sum of two graph instances. The sum of two graphs \( g_1 \) and \( g_2 \), denoted as \( g = g_1 + g_2 \) takes a pair of graph instances \( g_1 = (v_1, k_1) \), and \( g_2 = (v_2, l_2) \) and computes a new graph \( g \) as follows: (1) the node set of \( g \) is computed as \( v = v_1 \cup v_2 \) where duplicates are eliminated, and (2) the link set of \( g \) is computed as \( l = k_1 \cup l_2 \) where duplicates are eliminated. Notice that in this definition, two graphs with no shared node types can also be summed together.

The graph cross-product operator on two sets of graphs, denoted as \( G_1 \times G_2 \) combines instance graphs from each set in a combinatorial fashion by computing the sum of graph instance pairs consisting of graphs from each graph set.

The graph join operator, denoted as \( \Join \), takes two sets of graph instances \( G_1 \) and \( G_2 \) and a join subgraph expression \( \theta \). The expression \( \theta \) specifies the subgraph which should be satisfied by a pair of subgraph instances to be considered for summation. A graph join returns the set of graph instances \( g = g_1 + g_2 \) that share instance of the expression \( \theta \). Notice that when \( \theta \) is not specified, the graph join degenerates to a graph cross-product.

Notice also that this definition of join is purely structural. If we want to specify join predicates over attributes of graph elements in \( \theta \), we can specify these predicates using selection on each joining graph.

It is important to note that graph join is different from concatenation of paths from two graph sets. Concatenation is a special type of graph join where the operand graph is just a path and the join expression consists of a single node which also should be the end node in each path. For a subgraph, path concatenation can be generalized to be a concatenation of each path in the subschema expression.

**Example 2** Over the schema given in figure 1, the query in figure 4 finds those researchers that are from institutes located in California and attended a conference outside USA.

### 4.2.4 Set Operations

As in the relational model, we resort to set theory for union (\( \cup \)), intersection (\( \cap \)), and difference (\( - \)) of graph sets. The major issue that is specific to applying such operations on graph sets is how to determine when two graph instances are identical (for duplicate elimination purposes). Two graphs are said to be equal if they have equivalent node and link sets. Formally, two graphs \( g_1 \) and \( g_2 \) are equal if there
exists an isomorphism \( i : G_1 \rightarrow G_2 \) between the sets of nodes that preserves edges, and every attribute value in the corresponding node or edge type are equal.

4.3 Grouping and Aggregation Operators

Unlike in the relational algebra, grouping and aggregation are separate operators in gAnalytic. In relational algebra, grouping partitions the relation into sets of tuples. In our case, grouping has its own role of restructuring subgraphs independent of aggregation operations. On attributed graphs, grouping and aggregation operations can be viewed as a kind of navigation through the graph where graph restructuring is performed by a grouping operator and computations over structural properties as well as graph element attribute values are performed by an aggregation operator. Hence, though separate operators, grouping and aggregation are expected to be used in an integrated manner in our algebra.

4.3.1 Groupby

The groupby operator takes a set of (possibly non-disjoint) attributed graphs as well as a set of grouping parameters as input and returns a set of (not necessarily disjoint) attributed graphs.

In addition to the operand’s schema expression \( S \), the groupby operator, denoted as \( gby_{\beta, \phi}(S) \), takes the following parameters:

- A grouping basis \( \beta \): is a subschema expression that is a subset of the operand subschema based on which the graphs are grouped. In its simplest form, the grouping basis can be either a node or link label or even an attribute of a node or link. Hence grouping can be done based on structural as well as attribute data.

- Equivalence function \( \rho \): is a function (predicate) on some attribute(s) of a node or link that determines when two instantiations of the grouping basis in the operand graph are said to match. If \( \rho \) is not given, the keys of all nodes and links in \( \beta \) are matched.

The semantics of grouping is defined as follows: on a set of graphs, find all instances of \( \beta \), perform matching over these instances using the equivalence function and combine all matching (compatible) instances into a new subgraph which will inherit all links to the grouped subgraphs. This operation essentially combines multiple disjoint or non-disjoint matching subgraphs into a connected graph. However, notice that a single graph may contain multiple instances of \( \beta \) and hence grouping can also be performed even on one graph instance alone. Also notice that the groupby operator can perform a grouping similar to the grouping operation in the relational algebra when the input expression contains only a single node type and the grouping basis is based on a subset of attributes.

When a graph element in the grouping basis has multiple instantiation in the input graph (for example, when \( \beta \) contains a recursive link and a closure is defined over it in the input expression), all same type nodes in the grouping basis that are equivalent under \( \rho \) and are adjacent to each other are grouped together.

4.3.2 Aggregation

The purpose of aggregation operator is to apply standard as well as user defined aggregation functions over the input graphs’ structure or, in some structurally constrained manner, over attribute values of graph elements. It is important to notice that, unlike in the relational algebra, aggregation can occur on a graph without a grouping operation preceding it (for example, aggregation on a node with multiple links of some type).

The aggregation operator, denoted as \( agg_{M, \phi, A}(S) \), takes the following parameters:

- the input (operand) subschema expression, \( S \).

- Mode of aggregation, \( M \): This parameter specifies the method of aggregation to be used. Below we discuss two basic modes of aggregation.

- Aggregation function, \( \phi \): This is the function that computes the aggregation. It is defined over a graph element or its attributes or some sub-expression of the input subschema. The common aggregation functions are MIN, MAX, COUNT, SUM, etc. Likewise, this function can also be some user-defined function. However, the particular function used is orthogonal to the algebra.

- Decoration target, \( A \): This specifies where the newly computed aggregated value should be inserted.

Next, we discuss two of the parameters in detail; the other two are self-explanatory.

Mode of Aggregation. We define two basic modes of aggregation based on the manner of navigation done through the graph to carry out the aggregation. These two modes of aggregation are called branch-wise and path-wise. For a given node, a branch-wise aggregation applies the aggregation function over all of its adjacent qualifying graph elements (i.e. instantiations of the nodes or links over which the aggregation function is defined). For a given node, a path-wise aggregation applies the aggregation function over a path in the data graph provided that each graph element
on this path is qualifying. Typically, a path-wise aggregation is chosen when a closure operator is specified over the qualifying graph elements (or sub-expression) in the input subschema expression.

It is possible to combine branch-wise and path-wise aggregations in a query. In fact, this combination is commonly employed to achieve a rich expressive power. For instance, a path-wise aggregation over a path expression, say \((t_1.e.t_1)\)*, can a nested branch-wise aggregation over nodes of type \(t_2\) linked to each instances of \(t_1\) in the path. As another example, using the path-wise aggregation, one can express queries that find the shortest paths between two node types in a given subschema expression.

**Decoration(adornment) target.** It is necessary to specify the graph element and its attribute that will store the aggregated value. In relational algebra, the aggregated attribute is simply appended as an attribute in the result relation. For aggregation over attributed graphs, it is important to explicitly specify where and how the aggregated value should be placed in the output aggregated (decorated) graph. The algebra does not impose any particular constraints on how this should be done; we leave this as an extensible feature. Basically, the decoration target can be one of the graph elements from the input graph or a totally new graph element attached to the input graph in a specific manner.

We define two basic types of decorations that correspond to the two types of outputs that can be produced by the aggregation function:

- **Aggregated node.** Aggregations that result in a node decoration (e.g. a real value). Examples of this include degree (count) of incident nodes, centrality, betweenness (flow betweenness), page rank, etc.

- **Aggregated edge.** Aggregations that result in a decorated edge (e.g. impact). This is also equivalent to a matrix or decorating a node with a vector of values. Examples for this include aggregated values pertaining to a pair of nodes like shortest path, dependency, connectivity (line, node), max flow, accessibility, effects, etc.

**Example 3** Figures 5 shows an example of a branch-wise aggregation that decorates the researcher node with the maximum year in which s/he published a paper in each conference. Notice that the same result can also be represented by decorating the research-conference link by a maximum year value. Figure 6 shows an example of path-wise aggregation and labels each paper with the min year of papers that are linked to it in any link type. Figure 7 depicts a simple example of path-wise aggregation that also creates a new aggregated edge type. The aggregated edge is labeled with the shortest path between every pair of nodes in the example graph.

5 Graph Analysis using Hierarchical Views

In this section, based on the algebraic operations defined in the previous sections, we define more operators that are specifically aimed at OLAP-like hierarchical analysis of attributed graph data. In almost any major data analysis task, hierarchies (e.g. temporal hierarchies, spatial hierarchies, concept/ontological hierarchies) play an important role. For example, hierarchies are widely used in OLAP and ontology based data retrieval and analysis. Hierarchies provide semantic structure to data and correspondingly enhance the class of interesting and meaningful queries one can pose against it. For instance, hierarchies can be used to specify the restrictions on the data as well as the level of aggregation.

With particular reference to graph data, hierarchies are useful when we want to analyze relationships and interactions among aggregated entities based on data from lower-level link information. For example, on our example data set, an application can be interested in decorating links among higher level node types, for example topics or conferences or institutions, by values like measures of strength of interaction (e.g. influence) that are computed based on information from interaction graphs of hierarchically lower granule node types, for example citation graphs. Yet another example from the same application is analyzing the collaboration strength among institutes based on co-authorship of faculty. A number of similar examples exist in a wide variety of applications.

In the following sections we define various operators that enable hierarchical analysis over attributed graph data. First we describe how hierarchies can be defined as views on top of an attributed graph database to be stored as part of the graph schema. Then, we discuss how the groupby operator can be used to instantiate these hierarchical views. Finally, we define the Roll-up and Drill-down operators inspired by their namesakes in the OLAP field.
Hierarchical views are defined over the graph schema and stored as part of this schema. Essentially, a hierarchical view attaches further semantics to links by imposing hierarchical semantics (interpretation) to the relationships. A hierarchical view definition can be composed entirely of nodes and links from the schema or introduce new node and edge types to store some aggregated values (see also section 5.3). However, the introduction of new node or link types has to obey some restrictions. Particularly, at least the lowest level node of the view should be a node from the schema.

Notice that nodes that are at the same level in a hierarchy can also have links among themselves. So, we can view hierarchical views as multiple attributed graphs at different levels of granularity that are stacked on top of each other. A node in this view has vertical links with its “hierarchical neighbors” and horizontal links with its same-level neighbors.

Based on the types of graph elements involved, hierarchies are divided into two basic types:

- **Polytype hierarchy**: is a hierarchical view in which a path over the node and link types consists of multiple node types. For example, the hierarchy institute — researcher — paper depicted in figure 8 is a polytype hierarchy. This type of hierarchy is commonly based on domain specific semantics of containment, specialization/generalization, etc.

- **Unitype hierarchy**: is a hierarchical view defined over a single node type. These hierarchies are usually defined over values of some attribute of the node or link type. Hence, such hierarchies are also referred to as "Attribute-value taxonomies" [18]. For example, a topic hierarchy defined over a papers in a citation graph, depicted in figure 8 is a unitype hierarchy. So is a hierarchy over a traffic network based on spatial regions.

5.2 Hierarchical Grouping

Given a hierarchical view definition, the groupby operator can be used to express it. In the algebra, a hierarchical view translates to performing a groupby over the subschema expression of the hierarchical relationship where the higher-level node is a grouping basis (the equivalence function being, by default, the grouping basis’ id).

**Example 4** For example the institute — researcher — paper hierarchy in figure 8 is expressed as \( gby_I(gby_R(gby_P(A.researcher,P.paper))) \) where \( P, R, \text{ and } I \) stand for paper, researcher and institute, respectively.

When a view definition includes a new node or link type, we need to include an expression to create the aggregated node or link. In case of a new node, for each group that results from a groupby operation, a new aggregated node is created and all the aggregated nodes are linked to it by the link type specified in the view definition. In case of a new link, a new aggregated link is created for each result group incident with the nodes specified in the view definition.

5.3 Graph Roll-up and Drill-down Operators

In the preceding sections, we discussed how hierarchies can be specified and instantiated over a graph database. Here we show how aggregation is performed over these hierarchies. Specifically, based on the aggregation operator \( agg \) defined in section 4.3.2, we define two operators, namely graph rollup, denoted as \( grollup \) and graph drilldown, denoted as \( gdrilldown \) that are used to propagate aggregated values over hierarchical views of attributed graphs.

Intuitively, \( grollup \) propagates aggregations up the hierarchical view while \( gdrilldown \) provides the opposite capability, propagating disaggregated values down the hierarchical view. The aggregation that are rolled up and down can be over some structural property (for example, degree count, path length, etc.) or over some attribute values of
Example 5 Consider the graph example show in figure 1 and the corresponding hierarchic views shown in figure 4.3.2. Suppose we want to compute some aggregation (specifically an influence metric) over the citation graph of papers and want to rollup this aggregated value over the hierarchy institute — researcher — paper. Take the formula for the particular influence metric, drawn from the social network field [15] to be \( s_i = e_i + \sum_{j=1}^{n} r_{ij} s_j \), where \( s_i \) and \( s_j \) are the status or influence of nodes \( i \) and \( j \), respectively, \( r_{ij} \) is some measure of strength to which \( j \) is influenced by \( i \), and \( e_i \) is an exogenous value. This formula can be expressed using our algebra as shown in figure 9. Let us designate the decoration of a paper node \( P \) with a hubbel’s score by \( P.h \). Then suppose that we want to equally distribute a paper’s status metric to its authors; that is the smaller the number of authors of a paper, the higher the proportion of the paper’s status each can get. For this we use the following aggregation functions, \( \phi_{\text{down}} \), \( P.h_R = \text{count}(R.P) \) over the hierarchy expression \( R.P \). \( P.h_R \) stores a paper’s status with respect to a particular researcher.8 Then, finally to compute a researcher’s status, we use the following aggregation functions, \( \phi_{\text{up}} \), \( R.h = \text{sum}(P.h_R) \) over the same hierarchy expression \( R.P \).

Rolling up aggregation values in the above example was complicated by the fact that the hierarchy may not always be a one-to-many relationship. That is, there can be a many-to-many relationship between two nodes in a hierarchy such that a “child” node is linked to many “parent” nodes and vice versa. In such cases, we need to define how aggregated values need to be propagated.

Now we define the \textit{grollup} operator. The \textit{grollup} operator takes the following parameters:

- \textit{rollup link}, \( \mathcal{L} \), a subschema expression containing a pair of node types in a hierarchic relationship (subset of the hierarchic view expression).

- \textit{Upward aggregation operator}, \( \text{agg}_{\mathcal{M},\phi,A}^{\uparrow}(\mathcal{L}) \): an aggregation function to decorate higher level nodes with aggregations over lower level nodes. This can involve a path-wise aggregation for each node or simple projection of node attribute values.

- \textit{Downward aggregation operator}, \( \text{agg}_{\mathcal{M},\phi,A}^{\downarrow}(\mathcal{L}) \): an aggregation function to decorate lower level nodes with aggregations over higher level nodes. This can involve branch-wise aggregation function.

The application of the upward and downward aggregation is order-dependent. Hence, the query should specify the desired order. Also notice that aggregations over edges as well as nodes can be rolled up a hierarchy (e.g. group multiple edges among papers into one edge among authors and roll-up, for e.g. count of edges, aggregation of some edge attributes, etc.). Which one is desired depends on what is specified in the aggregation function \( \phi \).

6 Related Work and Comparison to Other languages

The literature is fraught with query languages that are developed for some type of graph model of data. Examples are GraphLog [9], Gram [1], GraphDB [13], GQL [2, 3], G-Log [17], Hydra [6, 4, 5], and recently Proximity [8]. A language aimed at extending SQL for querying graph relations is also presented in [7]. In general, this body of work differs from our work in two main ways: the data model and the type of operators defined. The data model of the above languages is limited to only edge labeled data where nodes contain atomic data values; some limited to rooted graphs where data is stored only on leaves; some limited to object-oriented view of links (i.e. inheritance, containment, etc.), etc. On the contrary, ours is based on the generic attributed graph model where both nodes and links have equal status and represent entities and their relationships. The nodes and links in our model also have arbitrary number of attributes associated to them.

Due to their sole focus of supporting simple, ad hoc queries over graphs, the above languages also presented some subset of the basic operators that performed selection, projection, joins and set operations on graph data. In fact, with the exception of Gram [1], none of the previous work presented a language that defined even the basic operators in their entirety. To the best of our knowledge, no previous work considered grouping and aggregation in an algebraic framework with the exception of [10] that discussed some aggregation features based on the GraphLog [9] language. Our language has full-fledged definition of grouping and aggregation. Furthermore, we define OLAP like operations on graph data.
7 Conclusion

In this paper, we introduced a minimal, yet powerful set of query operators as part of a graph query language called gAnalytic. Our proposed operators are composable, reorderable, and closed in application. They make possible the declarative specification and optimization of complex analytical queries over attributed graph data. These operators have been designed to be translated to SQL and can be implemented on top of an object-relational database system. In effect, they provide an algebraic application programming interface (API) that allows the separation of the front-end from the back-end. Finally, the proposed model provides a framework in which to study the efficient execution and optimization graph query components of various graph analysis and mining algorithms and techniques.

8 Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0083489 and by the Knowledge Discovery and Dissemination (KDD) Program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References


