Business Intelligence on Complex Graph Data*

Dritan Bleco
Athens University of Economics and Business
76 Patission Street
Athens, Greece
dritanbleco@aueb.gr

Yannis Kotidis
Athens University of Economics and Business
76 Patission Street
Athens, Greece
kotidis@aueb.gr

ABSTRACT

Advances in the Internet of Things will provide massive amounts of context-aware information that would need to be ingested and understood by supporting IT infrastructures, influencing running processes that trigger actions. As a result, future Business Intelligence (BI) platforms would need to be able to process and analyze complex data. In this paper, we adapt a generic graph model that may be used to represent data in many applications of interest. We then show how analytical queries over these data can be naturally expressed via an OLAP-like aggregation framework we introduce. We describe how ad-hoc aggregations can be easily decomposed into smaller independent computations via a proper query rewriting mechanism. Our techniques provide the basis for selecting materialized views in order to expedite computation of frequent analytical queries in large datasets, enabling data warehousing of collections consisting of millions of graphs. Our experiments demonstrate the benefits of our methods.

1. INTRODUCTION

The vision behind the Internet of Things (where "things" refer to the general idea of arbitrary data acquisition infrastructures) is to provide management, scalability and heterogeneity of devices (sensors, RFID enabled objects, actuators) and users (humans but also machines). All these "things" will provide massive amounts of context-aware information that would need to be ingested and understood by supporting IT infrastructures, influencing running processes that trigger actions. Real-Time Enterprise (RTE) promises to leverage this type of data gathering technology to reduce the gap between when data is recorded in an organization and when it is available for information processing and decision-making. Nevertheless, adapting the RTE paradigm requires significant commitments, not only in deploying low-level sensing and data gathering infrastructures, but also in overhauling existing Business Intelligence (BI) platforms so that they can cope with complex data of disparate formats [1].

*This work was partially supported by the Basic Research Funding Program, Athens University of Economics and Business.

Complexity in the data may be structural or contextual. Structural complexity refers to multiple linked pieces of data (as in web logs), while contextual complexity arises from interdependencies of the data stream to its environment whether the later is real (e.g. in a supply-chain environment, context is provided by the spatio-temporal coordinates of the data), or virtual (e.g. the business process, in a service provisioning application). Thus, unlike retail industries, which mainly deal with flat transactional "basket-type" data, the future will call for decision making on data with complex internal structure. Recent IT platforms, such as Workflow Management Systems (WMS) and Customer Relationship Management (CRM) software already face the need to handle data that does not conform to the basket paradigm, primarily used in data warehousing. Instead, every "record" in these applications describes a sequence of events that captures the interaction between a customer's activity and various components of the organization. Such records are better described using a graph model that captures state transitions and related cost information or other business- or process-specific attributes.

Another application example includes sophisticated Supply Chain Management (SCM) that poses complex decisions for companies in competitive industries. Difficulty increases because of the need to draw on multiple information sources, especially when those sources provide asynchronous updates. RFID readers, placed at warehouses, trucks and distribution hubs capture the list of RFID tags sensed in their vicinity and, at the same time, record context information such as time of observation, location, environmental conditions (temperature, humidity) etc. In this example, the distribution network can be abstracted as a graph, while article distribution records (composed by multiple RFID observations obtained at different times and locations) are annotated sub-graphs of the network streamed by the monitoring infrastructure. Additional examples of applications that need to handle complex (structurally or contextually) data include Enterprise Asset Management (EAM) (where heterogeneous sensor-nets are used in monitoring critical infrastructures), factory automation, data analysis of web logs etc.

A common challenge in all aforementioned applications is how we model data that carries complex structural or contextual information and, subsequently, how we analyze massive volumes of such data in an effective manner? In order to address shortcomings of existing BI platforms, in this work we adapt graph models as a generic tool for describing complex datasets. We then examine the necessary primitives that can help decision makes analyze large volumes of graph data. Past work has demonstrated that conventional data warehouse implementations are incapable of providing flexible analysis of graph datasets [2, 3]. In the popular multidimensional approach proposed for the basket-data paradigm, data are projected and aggregated using a set of dimensions (e.g. prod-

ucts and customers). In a graph database the structural properties of the graph are also dimensions of interest.

In this paper we consider the problem of enabling BI analyses over large-scale graph datasets. We describe a comprehensive framework for modeling analytical queries that range over the structure of the graph. For example, queries like "find the shortest delivery path" in a SCM application are easily captured by our framework by first specifying a structural condition that limits the scope of the examined records (i.e. the subgraph of interest) and then describing a pair of aggregate functions that will be used to consolidate the collected measures (timing information in this example). The first aggregate function, termed *intra-path*, aggregates selected measures over a path (for instance the $SUM()$ function is used to compose the length of each path in the aforementioned query). The second function termed *inter-path* aggregates data over multiple paths given the structural conditions specified. In this query, $MIN()$ will be used for inter-path aggregation in order to compute the shortest path.

As will be explained, in our framework we can decompose complex, ad-hoc aggregations on graph data into smaller, independent computations via proper query rewriting. In the context of a large-scale data warehouse, our framework allows re-use of precomputed materialized query views during query evaluation and, thus, enables view selection decisions that are of immense value in optimizing heavy BI workloads [4, 5]. The contributions of our work are:

- We present a comprehensive framework for modeling and analyzing graph databases. We demonstrate that our framework allows the composition of complex aggregations on selected parts of the graph, allowing us to accommodate many interesting analytical tasks in real-life applications.
- We discuss how complex queries can be rewritten via a decomposition of the graph model. These rewritings enable graph-aware optimization of user queries through the use of precomputed materialized views.
- We provide an evaluation of our techniques using graph databases of realistic sizes. Our results demonstrate that expensive queries can be rewritten so as to re-use precomputed results selected by a view selection algorithm. Per these rewritings, the cost of a user query is reduced to a fraction of the cost of the original query that is oblivious to the existing materialized views in the system.

The rest of this paper is organized as follows. In Section 2 we present a motivational example, including analytical tasks that can benefit from our techniques. In Section 3 we discuss related work. In Section 4 we introduce our graph model, while in Section 5 we discuss analytical queries over the graph data, query rewriting and view selection. Section 6 presents our experiments and Section 7 contains concluding remarks.

2. MOTIVATION

As a motivational example we will describe a SCM application. This application tracks the different routes that a customer order follows from production lines to the consumer hands. A customer order is formed from products that may be produced at different product lines. Products follow different paths until they are delivered to the customer. Multiple warehouses are located among the production lines and the shipping points and can stage the products while the order is being assembled. At every location, RFID readers are used to keep track of the location of the articles. These RFID readers are part of our application and the data they produced is streamed to a central location for further analysis.

As we mentioned an order follows one or more paths so our web supply chain application produces graph data. Consider the graph depicted in Figure 1. The graph consists of a set of nodes and edges among them. The two nodes at the left (A and B) are product lines, while the right-most nodes (L, O and M) depict customer end-points (or pick-up locations, stores, etc.). All other nodes depict warehouse locations where the products are stored temporarily until they are delivered to the customer. The nodes may have several attributes such as location, storing capacity, etc. Similarly, customers have additional attributes such as customer name, customer category, etc. A decision maker would like to analyze data according to all these attributes over different parts of the graph.

Figure 2 captures information related to a particular customer's order that we will refer to as a *record* henceforth. For this order, different products depart production lines A and B and traverse warehouses that are in various regions ($D \ldots K$). The order completes when all articles have arrived at the customer end points L, M. In this example, we record as a single measure in each edge the number of days for transportation from the starting node of the edge to the ending node. As an example, in this record, articles produced at node A took 21 days to arrive at C. On the other hand, when products arrive to a warehouse location, they can be stored/delayed for several days. This wait-time is depicted as a measure within the node. For example, warehouse D induces a wait-time of 5 days for every product that arrives for this order (as will be apparent our framework can also describe the internal processing of products at node D in more detail, if this is required by the application).

In this setting, the sum of measures (including measures on nodes) along a single path computes the total time for products that follow this route, to arrive at the customer location. As an example the products that traverse locations along path $[ADIM]$ have a aggregate time cost of 61 days.

Assume that each node (based on other attributes we maintain in the data warehouse) is located in some region. The nodes in the dark area of Figure 2 are located in Athens. An analyst may want to abstract all nodes inside Athens and see them as a single node. This operation is analogous to a *roll-up* based on location attributes in the data cube terminology. This abstract node (lets call it U) has six input edges (edges that emanate from a node outside this region) and five output edges. Inside the abstract node there are internal paths (for instance $[DGK]$) and time costs associated with these internal paths.

Given these data there are a few basic queries that one would like to pose like:

- Q_1: What is the total order completion time?
- Q_2: What is the total processing time for parts that are shipped through warehouses located in Athens?
- Q_3: Which are the fastest and the slowest routes that connect a location where articles arrive in the Athens region and a location from where articles depart the Athens region?

Query Q_1 in this example requires us to compute the *longest path* between nodes A,B and L, O, M. This longest path computation will be performed in the graph instances of all orders stored in the data warehouse. Obviously this computation will require substantial processing, for an SCM application that keeps track of thousands of orders. For the sample data record on Figure 2 the longest path is $[BNM]$ with a delay of 122 days.

Query Q_2 is similar, however, this time the *longest path* computation needs to be restricted to consider only paths that transverse

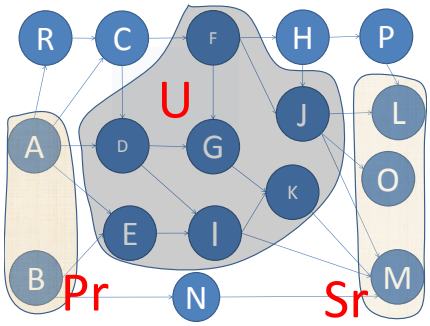

Figure 1: Example of a schema graph: Pr (A,B) are the production lines;Sr (P,L,O,M) are the customer end points, the rest of the nodes depict warehouses. The dark area represents locations within the Athens region

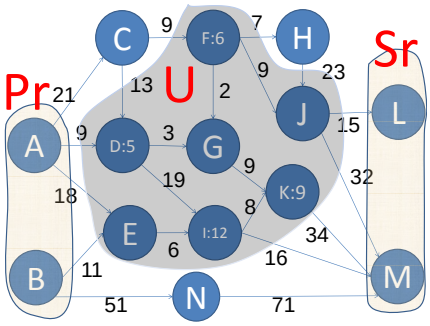

Figure 2: A subgraph (record) with associated measures

at least one location in Athens. The result of the Q_2 query for this record is path $[ADIKM]$ with an aggregated cost of 96 days.

Query Q_3 calculates the *longest and shortest paths* among routes with a starting and ending location at the borders of the Athens region. As an example, warehouse F depicts a route ($[FF]$) with an aggregate time of 6 days. In this record, the longest route is $[DIK]$ with a value of 53 and the shortest one is $[FF]$ with 6.

Our framework allows us to model such queries in an intuitive manner via a decomposition of a record into edges, paths and sets of paths (termed composite paths). We also provide for flexible aggregation of nodes in the graph that enables composition of nodes into aggregated ones (e.g. use U as an aggregate node that includes all Athens' locations). This way, we can express computations such as the longest path in a natural OLAP manner. We note that our abstraction is orthogonal to the way data records are actually stored and queried. For instance, in our implementation used in our experimental evaluation, we utilize a relational back-end in order to store and query graph records, however this choice does not influence our model. As will be explained, by allowing us to express interesting OLAP aggregations over graphs, our formalization provides a framework that we use to expedite costly queries via query rewriting and available precomputations in the data warehouse. For instance, assume aggregated (longest path) information on the movement of articles within the Athens region is pre-computed at a per-order basis and stored in the data warehouse as a materialized view (details will be given in the following Sections). Then, this view can be used to expedite processing of all queries we discussed, by reusing precomputed path aggregations and reducing the run-time calculations to locations outside Athens (when necessary, depending on the query)[1]. Thus, in the presence of millions of records in the data warehouse, the use of this view may have a profound effect in boosting query performance.

In order to effectively utilize precomputations via materialized views two important questions are raised:

- Given a large dataset with millions of graph records how do we select the proper set of views in order to expedite frequent computations?
- Given an ad-hoc user query and a set of materialized views in the data warehouse, how does one automatically derive the best rewriting of the user query that minimizes query execution time?

In what follows, we provide a systematic answer to both questions. We provide a methodology for rewriting a user query in a comprehensive manner and for a variety of interesting functions. Our framework enables selection of views for inclusion in the data warehouse and automated rewriting of a user query using these views.

3. RELATED WORK

Because of the size of the data and the (often) ad-hoc grouping and aggregation required, data warehouses rely on extensive pre-computation in the form of materialized views containing frequently asked aggregates. Engineering questions, such as how many and which views to materialize under a space and/or update time constraint and an expected query workload have led to several view selection algorithms [5, 6, 7].

In this paper we present an OLAP-oriented framework for analyzing data that contains both structural as well as measure information, in addition to other existing dimensions in the data. In database literature there is substantial work on recursive queries (e.g. [8, 9, 10]), which have motivated our work. The work of [11] first discussed measure aggregation over simple path expressions in a graph. In this work, we present a novel decomposition of the graph via the use of composite paths that naturally captures many interesting analytical tasks that are not addressed in [3, 11]. Recent work on materializing shortcuts [12] in RDF databases also assumes an underlying graph model for representing the data, however the particular query mechanisms involved are substantially different.

OLAP processing in graph data has been recently discussed in [13]. A key difference is that, in our work, we assume that (in addition to other recorded information), graph data is also recorded as facts in the data warehouse schema. Thus, there can be thousands or millions of such graphs that need to be analyzed. In contrast, the framework of [13] assumes a linked set of tuples that are described via a graph model (i.e. a single graph). Additional graph summarization operators are discussed in [14, 15]. Such techniques are complimentary and they may be used in conjunction to our framework.

4. GRAPH DATA MODEL

4.1 Preliminaries

In our work, we assume that the data records are graphs annotated with measure information related to the analysis we would like to perform. In order to provide a "schema" for these records,

[1]Whether this run-time computation is done in SQL or by executing an efficient graph algorithm after the records have been loaded in memory is orthogonal to our scheme.

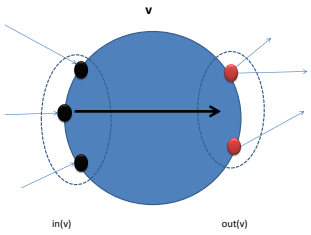

Figure 3: Abstraction of a node with an (unknown/hidden) to the application internal structure, depicted as an internal edge

we assume that there is an underlying directed *schema graph* $G_{schema}(V, E)$ that describes the set of possible nodes and edges. For example in the SCM application discussed in the previous section, the nodes in set V may describe locations where RFID readers are placed (production lines, warehouses, stores). Then, the edges in set E, describe possible routes for articles in the supply chain. For ease of exposition, we initially assume that the schema graph is acyclic. We will later remove this restriction. We also note that in addition to provide the schema for the data records, the schema graph may also be used in order to clean spurious incoming data, as RFIDs frequently generate erroneous observations [16].

In a typical data warehouse it is common that dimensions exhibit hierarchies that allow us to model and analyze records at different levels of granularity. Hierarchies are also evident in graph data. For instance multiple nodes of the schema graph that describe stores located within the same city may be aggregated in a "super node". Nodes may also be aggregated using ad-hoc conditions based on other dimension attributes (for instance the type of the store). In order to enable dynamic grouping of graph elements in an ad-hoc fashion, we introduce in our framework the concept of aggregate nodes. An aggregate node u is a connected subgraph G_u of the schema graph. For instance, in Figure 1 the dark area depicts an aggregate node for all warehouses located in Athens.

Given an aggregate node u, we use $in(u)$ to describe the set of nodes of G_u that have at least one incoming edge from nodes in G_{schema} that do not belong to G_u. Similarly, let $out(u)$ denote the nodes in G_u that have at least one outgoing edge towards a node in G_{schema}. In the example of Figure 1, $in(U)$=$\{D, E, F, J\}$ and $out(U)$=$\{F, I, J, K\}$. The set of nodes in $in(u)$ and $out(u)$ help us capture the connectivity of the aggregate node with the rest of the schema graph. Depending on the aggregation performed, it is possible that a node belongs in both sets, as is the case of nodes F, J in this example.

For completeness, we also let functions $in()$ and $out()$ range over regular schema nodes as well. In particular, for node v, $in(v)$ denotes the locations (within the scope of the node) where incoming edges arrive and, similarly, $out(v)$ denotes the locations where outgoing edges depart from v. Since G_{schema} determines the most fine-grained level at which information is depicted, the details of $in(v)$ and $out(v)$ are not known, as shown in Figure 3. However, this abstraction is also beneficial for two reasons. First, if the analyst decides at a later point to model the internals of node v in more detail, then she can simply replace v with a subgraph that will be added to the schema graph, without necessitating other changes to the schema. In addition, even if the internal structure of node v is not revealed to the decision makers, we can use an *internal edge* from $in(u)$ to $out(u)$, denoted as $(in(u), out(u))$ to capture measure information generated from processing within the node.

For instance, in the SCM record of Figure 2, node D may indicate a warehouse location with multiple platforms where articles arrive and depart, equipped with RFID readers. Additional readers may record the presence of articles at different areas inside the warehouse. The SCM application may decide that the details of the internal movement of articles in the warehouse is of no interest for the analysis. Thus, the warehouse is abstracted as a single node D, and the internal edge $(in(D), out(D))$ is used to capture the fact that articles for this record remain within its premises for a period of time (6 days in this example).

4.2 Path Types and Operations on them

In our analysis we would like to compute interesting aggregations over parts of the data graph, with respect to measurements collected at individual records. Our model uses an extended notion of a path in order to restrict attention to a particular part of the schema. A path in a graph application is simply a sequence of nodes resulting from the concatenation of adjacent edges. For instance $[A, D, G, K, M]$ (or $[ADGKM]$ for brevity) is a path whose constituent edges are (A, D), (D, G), (G, K), and (K, M). When a path is uniquely identified by its endpoints, for brevity, we omit the internal nodes. For example path $[A, C, F, H]$ is depicted as $[A, H]$.

In our setting, where nodes have internal (known or hidden) structure as well, additional formalism is needed. For example assume we would like to concentrate in the movement of articles after they depart warehouse F, up to the time they enter warehouse location K. Thus, internal measurements on nodes F and K should be left out of the analysis. In our terminology, this path is defined as starting from $out(F)$ and ending in $in(K)$. Borrowing notation from mathematics, we denote this "open-ended" path similarly to an interval whose endpoints are excluded: (F, K). When two nodes are connected via an edge, as in the case of A and C, the open ended path (A, C) is naturally mapped to edge (A, C).

Similarly, a path can be opened in only one of its side nodes. For instance path $[F, K)$ indicates a path that describes movement of articles from the time they enter warehouse F up to the point they enter warehouse K. Thus, internal measurements at F are also included, unlike the previous example.

Quite often, analytical tasks involve multiple paths from the schema graph that have the same starting and ending nodes. For instance an analyst may want to analyze the movement of articles from location A to location L. We observe that the schema graph contains multiple paths that connect these nodes. For brevity we will use the notation $[A, L]^*$ to denote the set of all these paths and will refer to it as a *composite path*. Composite paths may also be open ended for example $[D, K)^*$. In what follows, we sometimes omit the asterisk when referring to a composite path and it is obvious from the context.

Notation is naturally extended to aggregate nodes. For instance recall that U is an aggregate node denoting all existing warehouse locations in Athens (Figure 1). Then, $[A, in(U))^*$ denotes all paths starting from node A and ending at a node in set $in(U)$. Thus, $[A, in(U))^*$= $\{[A, C, D), [A, C, F), [A, D), [A, E)\}$. Similarly, composite path $[in(U), out(U)]^*$ can be used to trace articles from the time they first enter a warehouse location in Athens up to the time they depart towards a customer location. The use of $in()$ and $out()$ function is also applicable to regular nodes. E.g. $[in(D), out(D)]$ denotes the node's internal edge. For brevity we omit the $in()$ and $out()$ functions i.e $[D, D]$=$[in(D), out(D)]$. Path $[D, D]$ is used to indicate interest in measurements collected internally at node D.

In order to allow composition of paths we further introduce the

path-join operator ($\bowtie$) that concatenates two paths p_1 and p_2 when the ending node of p_1 is the same as the starting node of p_2 and one of the two paths is open-ended at the common end-point. For example $[A, C, F) \bowtie [F, H)$=$[A, C, F] \bowtie (F, H)$=$[A, C, F, H)$. On the contrary, path $[A, C, F]$ does not "join" with $[F, H]$ since they both include node F and the resulting composition is not a path (the internal edge $[F, F]$ would be repeated otherwise). The operator is applied to composite paths as well by considering path-joins among all pairs of paths in them.

The path-join operator allows us to express queries over the schema graph. For instance, if we are only interested in articles that pass through warehouses in Athens, we can indicate all relevant paths using expression $[Pr, in(U)) \bowtie [in(U), out(U)] \bowtie (out(U), Sr]$. Informally, this decomposition states that articles departing from Pr reach a warehouse in $in(U)$, then there is some internal processing within U and finally they are shipped to Sr via a warehouse in set $out(U)$. We node, that the aforementioned expression does not include path $[B, N, M]$ as the latter does not contain any location in Athens.

This decomposition of path expressions, made possible by our formulation, will be the key in optimizing complex aggregations over the schema graph, as will be explained.

4.3 Data Records and Operators on them

In our framework we do not restrict ourselves to a particular physical implementation of the database. For instance, the relational schemas for graph datasets discussed in [3] are applicable to our setting. We will thus describe our data in an abstract model that is independent of the storage implementation. A data record is a subgraph of the schema graph where edges (including internal edges) are annotated with measures we would like to analyze. For ease of exposition, we will only refer to examples that contain a single measure per edge, however, our framework is still applicable when multiple measures are recorded. Figure 2 presents an example of a record in the SCM application. We can see that the record contains measurements at instances of the schema edges and, additionally at internal edges, i.e. (F, F), (D, D), (I, I), and (K, K).

In order to decompose a data record into its constituent paths we define the *path-projection* operator π_p that projects the record on the edges defined in path p, while retaining their measures. For instance, if we would like to analyze the shipment of articles (described in the record r of our running example) from location A to C and then until they leave warehouse F, we use $\pi_{[A,C,F)}(r)$=$\{(A, C) : 21, (C, F) : 9, (F, F) : 6\}$. We implicitly use $(e) : m$ to indicate that measure m is associated with edge e.

The projection of a record on a composite path is computed as a set containing the projections into the constituent paths. Obviously, some of these projections may return an empty set, for records that do not contain all constituent paths. As an example:

$$\pi_{[AD)^*}(r) = \{\pi_{[A,D)}(r), \pi_{[A,C,D)}(r), \pi_{[A,R,C,D)}(r)\}$$

$$= \{\{(A, D) : 9\}, \{(A, C) : 21, (C, D) : 13\}\}$$

5. BUSINESS INTELLIGENCE ON GRAPH DATA

5.1 Foundational Computations

The framework we introduced in the previous section allows us to perform ad-hoc selection of paths and associated measures in a data graph. We now discuss how we can perform analysis on the selected data. Computations based on additional dimensions that are stored in the data warehouse (e.g. selection of a subset of graph records based on the type of order, the date etc) are orthogonal and may complement our techniques.

As explained, the projection of a record on a path p returns the set of edges belonging to that path along with their measures. We can, then, use an *intra-path* aggregation function in order to compute interesting statistics on these measures. Formally, an *Intra-Path Aggregation* function $F_p(r)$ takes as input a path p and a record r. The function F is applied on the measures resulting from the projection of record r on path p. The function returns the path p along with the computed aggregate.

As an example, in the record of Figure 2, assuming the $SUM()$ function is selected, $SUM_{[A,C,F]}(r)$ returns path $[A, C, F]$ and its duration (length), i.e. 36.[2]

Intra-path aggregation is also performed on composite paths. Recall that a composite path is a set of simple paths. In the case of composite paths the function is computed over the (existing) individual paths and returned along with the path. For instance,

$$SUM_{[AI]^*}(r) = \{SUM_{[A,D,I]}(r), SUM_{[A,E,I]}(r), SUM_{[A,C,D,I]}\} =$$

$$\{[A, D, I] : 45, [A, E, I] : 36, [A, C, D, I] : 70\}$$

Similar to the notation used for edges, we use p:v to indicate that the aggregation on path p returns value v.

Inter-path aggregation may be used in a subsequent step in order to further consolidate the results obtained via intra-path aggregation. As an example, function $MAX(SUM_{[Pr,Sr]}(r))$ computes the order completion time for the order depicted in record r. The intra-path function computes the length of every path from a production line node in set Pr to a customer location node in set Sr. The inter-path aggregation function MAX returns the maximum of these values, which denotes the order completion time. For functions like MAX, MIN the function may also return the path with the maximum (respectively minimum value). Thus, query Q_1 is expressed as ($\mathcal{R}$ is the set of all records in the data warehouse):

$$Q_1 = \{r, MAX(SUM_{[Pr,Sr]}(r)), \forall r \in \mathcal{R}\}$$

Query Q_2 is more demanding. Recall that we can enumerate all paths via a warehouse in Athens using expression $[Pr, in(U)) \bowtie [in(U), out(U)] \bowtie (out(U), Sr]$. The first path-join operator concatenates "external" paths towards a node in $in(U)$ with paths that are internal in aggregate node U. Similarly, the second path-join extends the resulting paths with routes towards a customer location in Sr. Thus, we can write query Q_2 as:

$$Q_2 = \{r, MAX(SUM_{[Pr,in(U))\bowtie[in(U),out(U)]\bowtie(out(U),Sr]}(r)),$$

$$\forall r \in \mathcal{R}\}$$

5.2 Query Rewrite

In the previous subsection we explained how queries of interests can be modeled in our framework. While, the details of how these queries will be expressed in the underlying implementation (for instance their SQL equivalent in a relational back-end) is orthogonal to our work, our abstraction is beneficial in that in facilitates rewriting of these queries. Thus, it can provide the basis for optimization decisions, including use of materialized views, as will be explained.

As an example, query Q_2 that computes the order completion time for articles that are shipped through warehouses in Athens is

[2]In a relational implementation a unique path-id p_{id} may be used to identify the path.

written using three composite paths $[Pr, in(U))$, $[in(U), out(U)]$ and $(out(U), Sr]$, as shown above. The aggregate function over a concatenation among these composite paths can be written also as a concatenation among the aggregate function results of these composite paths. Thus, we have

$$Q = MAX(SUM_{[Pr,in(U))\bowtie[in(U),out(U)]\bowtie(out(U),Sr]}(r)) =$$

$$MAX(SUM_{[Pr,in(U))}(r) \bowtie_{SUM} SUM_{[in(U),out(U)]}(r) \bowtie_{SUM}$$

$$SUM_{(out(U),Sr]}(r))$$

Recall that the result of intra-path aggregation is a path associated with an aggregate measure. In this case, the path-join operator concatenates paths with common ending and starting nodes and, additionally consolidates their measures. In this example, we need to add (via function SUM) their measures, and this is indicated in the formula underneath the operator.

In our example we have $SUM_{[Pr,in(U))}(r) = \{[ACD) : 34, [AD) : 9, [AE) : 18, [BE) : 11, [ACF) : 30\}$, $SUM_{[in(U),out(U)]}(r) = \{[DGK] : 26, [DIK] : 53, [DI] : 36, [EI] : 18, [EIK] : 35, [FF] : 6, [FJ] : 15, [FHJ] : 36, [FGK] : 26\}$ and $SUM_{(out(U),Sr]}(r) = \{(JL] : 15, (JM] : 32, (KM] : 34, (IM] : 16\}$

Thus, Q = $MAX(\{[ACD) : 34, [AD) : 9, [AE) : 18, [BE) : 11, [ACF) : 30\} \bowtie_{SUM} \{[DGK] : 26, [DIK] : 53, [DI] : 36, [EI] : 18, [EIK] : 35, [FF] : 6, [FJ] : 15, [FHJ] : 36, [FGK] : 26\} \bowtie_{SUM} \{(JL] : 15, (JM] : 32, (KM] : 34, (IM] : 16\})$ =$MAX(\{[ACDKGKM] : 94, [ACDIKM] : 121, [ACDIM] : 86, [ADKGKM] : 69, [ADIKM] : 96, [ADIM] : 61, [AEIM] : 52, [AEIKM] : 87, [BEIM] : 45, [BEIKM] : 80, [ACFJL] : 60, [ACFJM] : 77, [ACFHJL] : 81, [ACFHJM] : 98, [ACFGKM] : 90\}) = \{[ACDIKM] : 121\}$.

In general, the rewrite for pushing intra-path aggregation on a path is of the form

$$G(F_{p=p_1\bowtie p_2}(r)) = G(F_{p_1}(r) \bowtie_H F_{p_2}(r))$$

where F,G,H are appropriate aggregate functions. This expression can be easily extended for the case of aggregate functions like AVG that are computed via simpler calculations (e.g. SUM,$COUNT$).

In the case of functions like MAX and MIN that return to the user one of their input paths, an additional rewrite is possible by pushing inter-path aggregation inside path-joins. When pushed inside a join, the function is calculated over paths with common starting and ending nodes. We use the duplicate elimination operator δ to indicate this behavior.

Thus, $MAX(SUM_{[Pr,in(U))}(r) \bowtie SUM_{[in(U),out(U)]}(r) \bowtie SUM_{(out(U),Sr]}(r))$ can be written also as
$MAX(MAX_{\delta}(SUM_{[Pr,in(U))}(r)) \bowtie_{SUM}$
$MAX_{\delta}(SUM_{[in(U),out(U)]}(r)) \bowtie_{SUM}$
$MAX_{\delta}(SUM_{(out(U),Sr]}(r)))$.

Intra-path aggregation with duplicate elimination selects one path per combination of starting and ending nodes, i.e. $MAX_{\delta}(\{[ACD) : 34, [AD) : 9\}) = \{[ACD) : 34\}$.

Continuing the calculation we get $MAX(\{[ACD) : 34, [AE) : 18, [BE) : 11, [ACF) : 30\} \bowtie_{SUM} \{[DIK] : 53, [FF] : 6, [EI] : 18, [EIK] : 35, [DI] : 36, [FHJ] : 36, [FGK] : 26\} \bowtie_{SUM} \{(JL] : 15, (JM] : 32, (KM] : 34, (IM] : 16\}$ =
$MAX(\{[ACDIKM] : 121, [ACDIM] : 86, [AEIM] : 52, [AEIKM] : 87, [BEIM] : 45, [BEIKM] : 80, [ACFHJL] : 81, [ACFHJM] : 98, [ACFGKM] : 90\})$= $\{[ACDIKM] : 121\}$.

Obviously the result of Q_2 is the same in both cases. The rewrite formula when pushing intra-path aggregate functions is:

$$F(G_{p=p_1\bowtie p_2}(r)) = F(F_{\delta}(G_{p_1}(r)) \bowtie_H F_{\delta}(G_{p_2}(r)))$$

5.3 Materialization of Frequent Calculations

A direct benefit of the rewrite rules we discussed in the previous section, is that we can use them to instruct the query optimizer of the underlying storage backend to consider pushing aggregate calculations under path-joins. Alternatively, this logic can be implemented at the OLAP front-end that is used to analyze the records. Rewrite rules can also help us boost query performance by exploiting materialization (pre-computation) of common (sub)queries. For instance, in our running queries, the database administrator may decide to materialize shared computations in queries Q_1 and Q_2 such as $MAX_{\delta}(SUM_{[in(U),our(U)]}(r))$, which consolidates paths within aggregate node U. If these calculations are stored in a materialized view (e.g. in the form of (record-id,path-id,measure) in a relational implementation), then at query time we can use the aforementioned rewrite rules to speed-up computation of both queries. Thus, we can revisit the view selection problem, in the case of aggregate computations over graph records in a data warehouse.

5.4 Discussion

In our discussion so far, we have assumed that the records we manipulate contain no cycles. This limitation is not inherent to our work. Consider for example Figure 1 and assume a back-edge (F, A) is added to the graph. In the SCM example this may indicate that violation of certain conditions (e.g. improper/unknown customer shipping address) results in returning the product back to node A. Consider now the following record.

$$\{(A, C) : 21, (C, F) : 9, (F, A) : 2, (A, C) : 20, (C, F) : 10, (F, F) : 6, \dots\}$$

Note that the record contains a cycle, specifically $A \rightarrow C \rightarrow F \rightarrow A$. Given this data, what is the intended meaning of an inter-path aggregation such as $SUM_{[ACF)}$? Assuming the recorded measure relates cost information, we can define the aggregation to include the summation of all instances of path $[ACF)$ in this record, i.e. (21+9)+(20+10)=60 or even include the measure of edge (F, A) in this calculation. Of course there may be other interpretations depending on the context. For instance, assuming the recorded measure captures the time it takes to transport an article and that we would like to find the time it takes for an item to first reach warehouse F, after it departs from A, then we can define a $SUM_{first\_occurrence}$ function to perform the calculations in the intended manner and return 30 in this example. Thus, in the presence of digraphs one needs to derive the proper aggregate functions, depending on the application.

Another extension to our framework is to consider micro-edges (and similarly micro-paths) within a record in order to facilitate analysis of multigraphs. A straightforward workaround is to consider the record as the union of multiple digraphs and use subsequent aggregation of micro-paths in the results of intra-path queries.

6. EXPERIMENTS

Our framework allows efficient execution of complex aggregate queries over graph data records, via the use to materialized views that contain pre-computed results of frequent computations. In what follows, we provide an brief experimental evaluation using the PBS (Pick By Size) algorithm discussed in [6]. This algorithm has been shown to perform well for traditional OLAP data. We also note that in the literature there are numerous view selection techniques that can be also adapted to our setting (for instance [5, 6]).

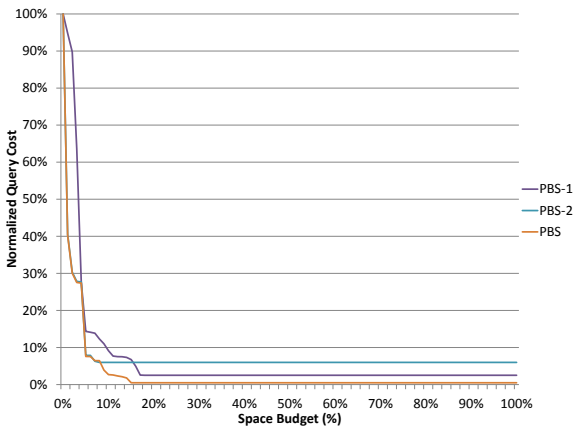

Figure 4: PBS, Bay Data Set, Uniform 100 Queries

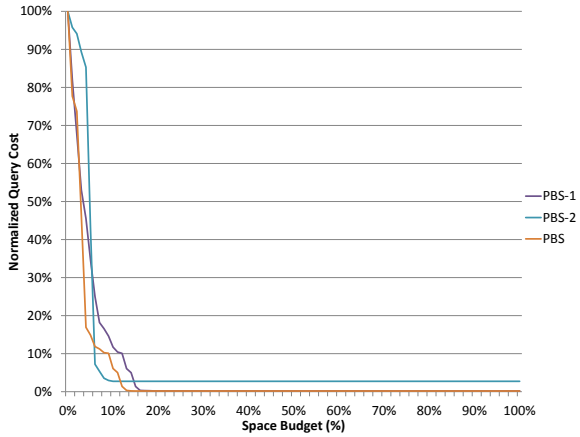

Figure 5: PBS, Bay Data Set, Zipf 100 Queries

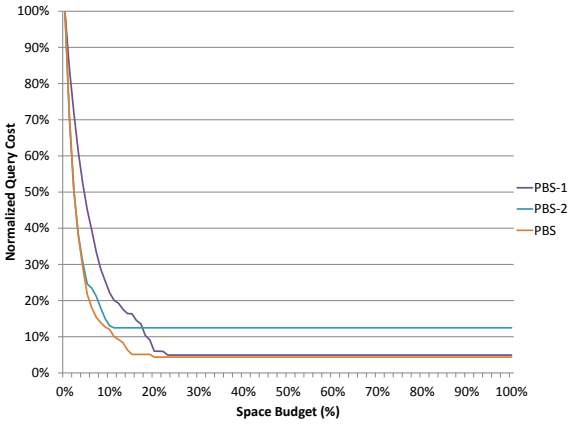

Figure 6: PBS, Gnutella Data Set, Uniform 100 Queries

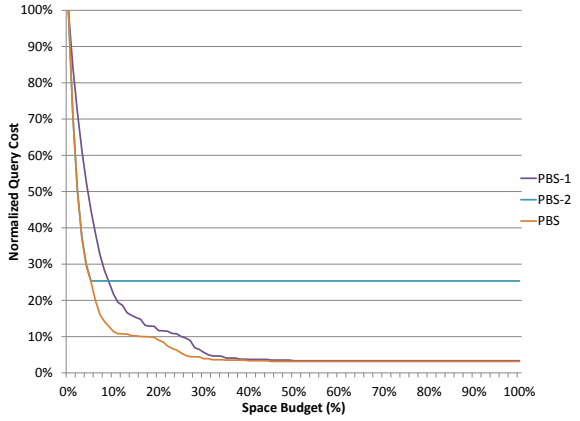

Figure 7: PBS, Gnutella Data Set, Zipf100 Queries

While these variations can also be considered (given our formulation of queries and rewrite rules), they are beyond the scope of this paper.

For our experiments we used the following two real Schema Graphs.

- **BAY:** This dataset depicts San Francisco Bay Area roads and was downloaded from: http://www.dis.uniroma1.it/∼challenge9/download.shtml. We selected a subset of this data that contains 9648 distinct nodes and 12000 distinct edges
- **Gnutella:** The second real schema graph describes connections among Gnutella hosts from August 2002. We used the full dataset which has 8846 distinct nodes and 31839 distinct edges. (For further information about this graph visit: http://snap.stanford.edu/data/p2p-Gnutella05.html.)

For each dataset we synthesized 120 million records and assigned random real values to the labels of each record. A record is a subgraph of the corresponding full schema graph. Our selection of datasets is justified as follows. The first dataset may be used to describe a distribution network within the city. Then, each record may depict the routes of one or more trucks for delivering a certain load. The second dataset depicts network traffic in the P2P network. A network administrator may use the recorded link usage information in order to calculate network utilization among different routes.

In our experiments we used queries that are chosen (either with uniform or with Zipf distribution) from the superset of all paths traversed by the records. Unless noted otherwise, half of them are intra-path and half inter-path aggregate queries. We used function $SUM()$ for intra-path aggregation, while $MAX()$ was used for inter-path aggregation. In all experiments we utilize a simple relational representation of the records as edge sets (each graph record is stored in multiple tuples, one per associated edge containing also the numerical measure). Aggregated measures for the materialized views are stored in separate tables. Of course, many different relational implementations of the records and the views are also possible and we expect their selection to influence quantitatively (but not qualitatively) the results we present in this section.

In order to obtain a platform independent evaluation, we calculate the cost of a query via the total number of tuples that need to be retrieved (having indexes on the record and edge-ids). The PBS algorithm takes as input the query workload and a space budget B that, in our experiments, depicts the maximum number of records that could be stored in the precomputed materialized views. We used three variations of the algorithm. The first algorithm considers only intra-path materialized aggregates and is denoted as PBS-1 in the graphs, the second (termed PBS-2) considers only inter-path

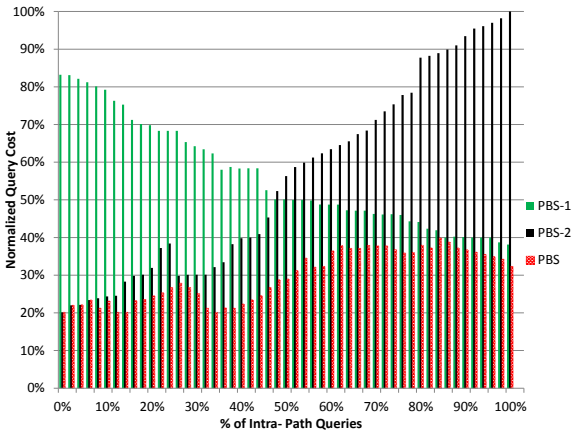

Figure 8: Varying Query Mix, BAY Data Set, Uniform Queries

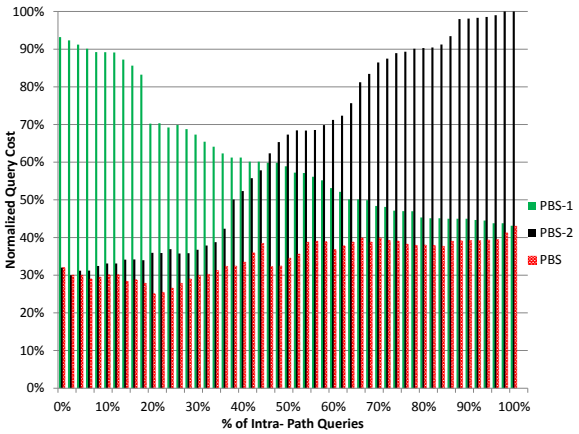

Figure 9: Varying Query Mix, BAY Data Set, Zipf Queries

aggregates, while the last (termed PBS) selects and materializes both types of views depending on the query workload.

Figures 4-7 depict the cost of query answering for 50 random queries (where selection of paths in the queries are generated according to the uniform and Zipf distribution, respectively) for both datasets. The y-axis depicts the normalized query cost, i.e the cost of executing the queries using the materialized views, over the cost of running the same queries without any rewriting. The x-axis shows the space budget as a percentage of the space that would be required if all queries were materialized in the system. Clearly all algorithms provide substantial benefits. For instance, with just 5% of the space required for materializing all queries, the query cost is reduced by a factor of up to 13. Comparing the full-fledged algorithm with the variants that restrict view selection to only inter-/intra-path types, we observe that it provides more benefits independently of the space budget used.

In the next experiment, shown in Figures 8-9, we vary the mix of intra-/inter-path queries in the BAY dataset for a fixed budget of 20%. Figures for the Gnutella dataset are similar and are omitted due to lack of space. When all queries are inter-paths (leftmost bars in the graphs) PBS and PBS-2 have the same performance, while performance for PBS-1 degrades. This is because, PBS-1 only considers intra-paths views that are of limited use for an inter-path query that needs to aggregate a large composite path. At the other end of the spectrum, when only intra-path queries are considered, PBS-2 cannot provide any benefits. Overall, PBS that considers both types of views provides consistently the largest reduction in query cost.

7. CONCLUSIONS

We have described a framework for modeling analytical queries in a graph database that is independent of the underlying storage representation of the records and the query language used. Our framework permits rewriting of complex aggregations into smaller computational units, enabling cost-based query optimization and pre-computation of frequently used calculations. Our experimental results validate our intuition that proper selection of materialized views can provide substantial gains in a large data warehouse containing millions of graph records.

8. REFERENCES

[1] M. Castellanos, U. Dayal, S. Wang, and C. Gupta, "Information Extraction, Real-Time Processing and DW2.0 in Operational Business Intelligence," in *DNIS*, 2010.

[2] J. Eder, "Extending SQL with General Transitive Closure and Extreme Value Selections," *IEEE Trans. Knowl. Data Eng.*, vol. 2, no. 4, 1990.

[3] Y. Kotidis, "Extending the Data Warehouse for Service Provisioning," *Data & Knowledge Engineering*, vol. 59, no. 3, December 2006.

[4] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing Data Cubes Efficiently," in *SIGMOD Conference*, 1996, pp. 205–216.

[5] Y. Kotidis and N. Roussopoulos, "A Case for Dynamic View Management," *ACM Transactions on Database Systems*, vol. 26, no. 4, 2001.

[6] A. Shukla, P. Deshpande, and J. F. Naughton, "Materialized View Selection for Multidimensional Datasets," in *VLDB*, 1998, pp. 488–499.

[7] D. Theodoratos, S. Ligoudistianos, and T. K. Sellis, "View Selection for Designing the Global Data Warehouse," *DKE*, vol. 39, no. 3, 2001.

[8] P. Larson and V. Deshpande, "A File Structure Supporting Traversal Recursion," in *SIGMOD Conference*, 1989.

[9] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan, "The Magic of Duplicates and Aggregates," in *VLDB*, 1990.

[10] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," in *SIGMOD Conference*, 1986, pp. 166–176.

[11] Y. Kotidis, "A Data Warehousing Architecture for Enabling Service Provisioning Process," in *VLDB*, 2001, pp. 481–490.

[12] V. Dritsou, P. Constantopoulos, A. Deligiannakis, and Y. Kotidis, "Optimizing Query Shortcuts in RDF Databases," in *ESWC (2)*, 2011, pp. 77–92.

[13] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu, "Graph OLAP: Towards Online Analytical Processing on Graphs," in *ICDM*, 2008, pp. 103–112.

[14] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient Aggregation for Graph Summarization," in *SIGMOD Conference*, 2008, pp. 567–580.

[15] P. Zhao, X. Li, D. Xin, and J. Han, "Graph Cube: On Warehousing and OLAP Multidimensional Networks," in *SIGMOD Conference*, 2011, pp. 853–864.

[16] S. Jeffery, M. Garofalakis, and M. Franklin, "Adaptive Cleaning for RFID Data Streams," in *VLDB*, 2006.