# Graph Analytics on Massive Collections of Small Graphs*

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

Emerging applications face the need to store and query data that are naturally depicted as graphs. Building a Business Intelligence (BI) solution for graph data is a formidable task. Relational databases are frequently criticized for being unsuitable for managing graph data. Graph databases are gaining popularity but they have not yet reached the same maturity level with relational systems. In this paper we identify a large spectrum of applications that generate graph data with specific characteristics that make them candidate for being stored in a relational system. We describe a novel framework where data and queries are both treated as abstract graph structures that can be decomposed into simpler structural elements. We complement this abstract framework with a description of a system that utilizes three different means of expediting user queries: (i) a flat description of the graph records using a column-oriented storage model, (ii) use of bitmap columns for enabling fast access to parts of these graph records and (iii) a novel framework for selecting and materializing graph views that significantly expedite retrieval of records in response to a graph query. To the best of our knowledge we are the first to report results using datasets consisting of hundreds of millions of graphs with billions nodes, edges and measure values using a single database server running of a commodity node. Our results demonstrate that our platform is orders of magnitude faster than alternative systems that natively handle graph data and a straightforward relational implementation. Moreover, our materialization techniques (that account for about 10% of extra disk space) are able to reduce the query execution times further, by up to 94% compared to an evaluation plan that is oblivious to the existing materialized graphs views in the database.

## 1. INTRODUCTION

The data management community has long been interested in problems related to modeling, storing and querying graphs [1, 2]. Recently, this interest has been renewed with the emergence of applications in social networking, location based services, biology and the semantic web where data graphs of massive scale need to be processed efficiently.

While such applications of large graphs are attracting a lot of attention, there are many other popular examples where smaller graphs are generated in a continuous basis. Examples include Customer Relationship Management (CRM) software, Workflow Management Systems (WMS) and Supply Chain Management (SCM) applications, all of which generate data records that can be naturally expressed as *graph records*. For instance, in a SCM application, a graph record is a graph that denotes the nodes (places) an article (or a collection of articles) has been, see Figure 1. The nodes and edges of this graph may contain interesting measures (e.g. timestamps, cost related attributes) that will be useful for analyzing the efficiency of the SCM application, detecting bottlenecks, etc.

While each of these records is typically small (e.g. the number of nodes and edges is in the order of hundreds/thousands and not millions as in a social network graph) and, thus, performance of atomic operations on them is not a concern, their cumulative size may easily overwhelm existing BI solutions, when complex analytical queries are considered. For instance, in SCM the number of graph records generated is expected to grow substantially with the increased adaptation of Radio Frequency Identification (RFID) technology and other sensory infrastructure. Other examples of applications that produce graph records include Service Provisioning and manufacturing execution using RFID tags [3]. A large-scale implementation of the aforementioned processes can easily generate millions of graph records on a weekly basis.

This shift from *flat* basket-type data to complex graph records necessitates reconsideration of Business Intelligence solutions. While graph and in general non-relational databases are attracting a lot of popularity, their maturity level is still questionable. A Relational Database Management System (RDBMS) constitutes a proven technology with very strong commercial and user bases. Are we ready to drop relational systems in favor of a new technology platform or should we try to adapt them to take on the new challenge? In this paper we make a bold statement that goes against popular belief. *We argue that relational systems can easily take on massive collections of graph records in the aforementioned applications, by utilizing and adapting techniques such as column-oriented storage, bitmap indexes and materialized views to the graph data model.* While these adaptations will be thoroughly discussed in the remainder of this article, we would like to emphasize that we do not claim that relational systems are best suited to handle arbitrary graph data and queries on them. The distinct difference between our targeted applications (e.g. Service Provisioning, Workflow Management, Supply-Chain Management) and other applications of graphs (e.g. social web, biology) is that (i) instead of a single massive graph our targeted applications need to manage a massive collection of smaller graphs and (ii) the nodes and edges of these smaller graphs

*This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: RECOST.

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

are named entities that are mapped to real world or business entities (e.g. a node may be mapped to a workflow state or a location) and not abstract elements. The second property implies that graph queries of the type of graph isomorphism that arise in e.g. computational biology and have been heavily studied in the past, are not the focus of our work.

Our framework puts together three different techniques in order to allow efficient processing of datasets consisting of millions of graphs with billions of nodes/edges. First, a column-oriented (yet still relational) storage is utilized so as to permit a flat description of the graph records. This simple but intuitive representation alleviates the known deficits of row-oriented relational systems when used for graph processing: recursion and the need to resort to costly joins for path calculations. Second, we build a very efficient indexing mechanism over this flat representation using bitmap columns on the column-store that are analogous to bitmap indexes frequently used in data warehouses [4]. Per the use of these bitmap columns, execution of complex queries is reduced to binary calculations on the stored bitmaps, enabling quick processing of complex graph patterns. Our modeling of these indexes is generic and can further accommodate specialized graph indexes (such as the gIndex [5]), if required. Finally, we complement our techniques with a framework that permits the creation of materialized graph views of different types. As will be explained, these views can be naturally incorporated in our flat data model and result in significant improvements in query times, especially for queries that perform aggregations on the stored measures.

In order to formalize better the types of analytical queries we consider, we complement our framework with a discussion of how complex data analysis tasks that are common in the aforementioned applications can be efficiently stated and executed. We describe a unified proposal where analytical queries on the records are themselves depicted as graphs that are optionally supplemented with aggregate functions denoting user-defined computations of interest. These *graph queries* consolidate measures associated with existing records. We explore techniques that break ad-hoc aggregations on graph records into smaller independent computations that we then expedite via the use of materialized graph views. While materialized views have been considered in data warehousing [6] or the Semantic Web [7, 8], the details of the views we consider, their selection process and usage in analytical queries over graph records are quite different.

We complement our work with a thorough evaluation of our techniques using graph databases consisting of hundreds of millions of graph records with tens of billions of nodes, edges and measure values on them, using a column-store running off a commodity node. Our results demonstrate that expensive graph queries can be rewritten so as to re-use precomputed graph views selected by our materialization algorithms. 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 graph views in the database. Moreover, we make direct comparisons of our techniques against: (i) a popular open-source graph database that natively stores and manipulates graph records, (ii) a commercial Resource Description Framework (RDF) store and (iii) a commercial RDBMS using row-oriented storage. These experiments highlight the effectiveness of our techniques for the data and queries we consider.

## 2. MOTIVATION

As a running example we consider data generated from an SCM application that utilizes sensory infrastructure (such as RFIDs) in order to trace delivery of articles produced by a large organization to its customers. A customer order is formed from articles that may be produced at different production lines. These articles often follow different paths through a network of service hubs before delivery. The collected trace data for a particular order may be visualized as a graph data record shown in Figure 1. In this graph record, nodes are used to represent locations of different types. The set of nodes on the left represent production lines, the middle nodes delivery hubs and the nodes on the right delivery points for this order. The edges of the graph depict delivery routes.

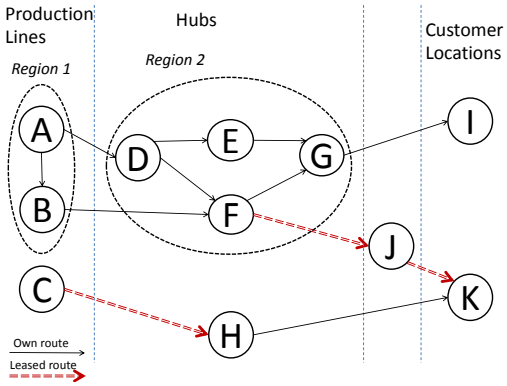



**Figure 1: A SCM Graph Record**

We assume that for this data two interesting measures are being collected: time and cost. These measures may be associated to edges, nodes or both. For example, the time measurement on edge $(A, D)$ denotes the time recorded for shipping the articles, for this order, from production line $A$ to hub $D$. The measures at node $D$ may denote internal delay or processing cost at this hub.

In our considered SCM application, graph records of this type are continuously generated from the monitoring infrastructure. There can be thousands of such records formed by the orders of a particular customer. Moreover, the delivery options may differ according to the customer type or order type (regular, fast-track, etc). Thus, there can be other locations or paths, not shown in the record above, which is only concerned with measured data for a single order.

Given this type of data, there are several computations that may be requested by an application that analyzes these records in order to detect anomalies or drive decision support tasks. Examples of such queries are

- $Q_1$: Compute the delivery time for all articles shipped via a certain delivery path, e.g. $[A, D, E, G, I]$ or a set of paths.
- $Q_2$: Assuming that delivery legs $[C, H]$ and $[F, J, K]$ are leased from a certain carrier, compute the delivery cost associated with shipment of articles via these routes.
- $Q_3$: Compute the longest delay for delivering an article of an order from a production line in region 1 to customer end-point $I$ via hubs from region 2.

Queries such as the above are common in SCM applications and raise several challenges. First, given a dataset consisting of millions of such graph records how to quickly locate records that obey ad-hoc structural conditions, such as the existence of one of multiple paths? Moreover, during a custom analysis, parts of these graphs may be consolidated based on user-selected conditions.

As an example in query $Q_3$, all production points within region 1 are consolidated into a single "aggregate node". Similarly all hubs within region 2 can be treated as a single aggregate node for this query. In the same spirit, node $H$ may correspond to a whole set of hubs and delivery routes within another region 3 whose details are hidden in this view of the data record because these details are of no interest to the application. Assuming that certain statistics on this hidden part, such as the overall delivery time and cost, are

pre-computed and stored along with the base records in the form of an aggregate node (view) $H$, we need mechanisms that will allow us to reformulate a given user query so as to exploit existing precomputations.

In this paper we provide a framework to address all these challenges. We formally define the notion of a graph query as the basic element for retrieving interesting measures and statistics from a collection of graph records. We then describe a query rewrite mechanism that allows us to re-formulate requested computations so as to utilize existing precomputed summaries in the form of graph views and we revisit the view selection problem in the presence of these new and challenging graph datasets.

# 3. GRAPH DATA AND QUERIES ON THEM

## 3.1 Graph Data Records

We assume that our dataset consists of a, possibly very large, collection of graphs annotated with measures of interest. Formally, a graph record is a directed graph $G_r(V, E)$, where $V$ is the set of nodes and $E$ the set of its edges. The nodes and edges of this graph are associated with one or more measure values that will be useful for analysis. For ease of presentation in what follows we assume that a single numerical value is associated to each of the elements (nodes and edges) of each graph record, however our techniques are applicable when multiple measures are recorded. In cases where a graph record contains cycles (for instance in the delivery record of Figure 1 the existence of a "back edge" from $D$ to $A$ may indicate that certain articles were damaged during shipment and returned back to the production line $A$), we adopt a simple flattening policy that assigns unique identifiers to the depicted nodes (for instance via a breadth- or depth-first traversal of the graph nodes). In this example we may use the following naming scheme and describe the corresponding edges as $(A_1, D_1)$, $(D_1, A_2)$, $(A_2, D_2)$ in order to remove the cycle on that part of the record. By enforcing the same naming scheme (for instance based on a breadth-first traversal of the nodes) on both data and queries (queries are discussed in Section 3.2), our techniques can handle arbitrary graphs.

We do not make any particular assumption on how these graphs are generated, thus, there is no requirement for modeling the schema of the underlying network (in a delivery service) or process (in a workflow management application) that generate this data. We only assume that the nodes are labeled using a universally adopted schema so as to be able to run queries on them afterwards by referring to common identifiers for the nodes. For instance a second graph record for a different customer may contain some of the nodes and edges depicted in Figure 1 and thus utilize the same identifiers annotated with its own data; or may contain other nodes/edges corresponding to a different transport network not associated with the one shown in this record.

Nodes and edges of the records may have additional metadata information (e.g. spatial location of a node, capacity of an edge, etc) that can be utilized by the application, however their presence does not affect our modeling. Similarly, in certain applications, a collection of graph records may refer to the same logical unit, as in the case where an order is broken into multiple sub-orders that are processed independently. This is also handled easily in our framework by using metadata information in order to relate multiple, basic graph records, for instance via the use of unique record-ids that join these sub-orders. The same logic allows us to handle cases where multiple edges need to be depicted (as in multigraphs), for instance in a parallel delivery of articles for a single order. All aforementioned scenarios can be handled via the use of multiple graph records linked together via additional metadata information.

Metadata on graph records are often utilized in order to form hierarchies of nodes and edges that allow us to analyze the underlying measurements at different granularity levels. For instance, in Figure 1 all nodes within region 2 may represent hubs located within the same province. For a certain analysis it may be more convenient to treat all these nodes as a single aggregate node and coalesce their measures along with the measures of the constituent edges accordingly. In our prior work [9], we have discussed operators that allow us to zoom-in/out of such ad-hoc groups of nodes in a formal manner and these extensions can be easily adopted to this work.

## 3.2 Graph Queries

Given that the datasets of our considered applications consist of graph records it is only natural to model our queries using the same foundation. Thus, in this work we describe our basic querying logic via graph models. A graph query $G_q(V, E)$ is a directed graph whose nodes are drawn from the same universe of nodes used in the graph records. The graph query indicates our intention to identify existing graph records that contain the same structural elements (nodes/edges) in order to retrieve and process their measures. Formally, a graph record $G_r$ is part of the answer of $G_q$, iff $G_q$ is a subgraph of $G_r$ and that answer consists of the graph $G_q$ annotated with the corresponding measures of record $G_r$. As an example, query $Q_1$ of Section 2 utilizes a single query graph

$$G_{q_1}(V = \{A, D, E, G, I\}, E = \{(A, D), (D, E), (E, G), (G, I)\})$$

in order to retrieve all graph records that contain the aforementioned path. For each such record, the query returns the corresponding subgraph (which in this example is a single path) along with its measurements.

Using multiple graph queries, we may form more complex logical conditions on the structure of the graph records under consideration. Let $[G_q]$ denote the set of records from our database that belong to the answer set of $G_q$. Then, we can easily derive formulas for computing basic logical operators between graph queries such as $[G_{q_1}\ AND\ G_{q2}]$=$[G_{q_1}] \cap [G_{q2}]$, $[G_{q_1}\ OR\ G_{q2}]$ = $[G_{q_1}] \cup [G_{q2}]$, and $[G_{q_1}\ AND\ NOT\ G_{q2}]$ = $[G_{q_1}]$ - $[G_{q2}]$.

Formulas such as the above can be used, for instance, in order to retrieve orders that deliver articles through hubs on region 2 but exclude hub $F$. When using bitmap columns, as will be explained in Section 4.2, for indexing the graph records, these formulas suggest that efficient retrieval of the requested graphs can be obtained via proper binary calculations on the content of these columns.

## 3.3 Paths: A Fundamental Structural Unit for Graph Queries

A graph query can be decomposed into one or multiple simpler queries that define paths of nodes. In this subsection we borrow the notation of [10] and use an extended notion of a path in order to restrict attention to particular parts of the records. A path is a sequence of nodes resulting from the concatenation of adjacent edges. For instance $[A, D, E, G, I]$ is a path whose constituent edges are $(A, D)$, $(D, E)$, $(E, G)$, and $(G, I)$.

When nodes have measurements associated with them, additional formalism is needed. For example assume we would like to concentrate in the movement of articles via node $E$, from the time they depart hub $D$, up to the time they enter hub location $G$. Thus, internal measurements on nodes $D$ and $G$ should be left out of the analysis. In [10], this "open-ended" path is treated similarly to a numerical interval whose endpoints are excluded: $(D, E, G)$. When two nodes are connected via an edge, as in the case of $D$ and $E$, the open-ended path $(D, E)$ is naturally mapped to edge $(D, E)$. Similarly, a node can be described as a path starting and ending on that node, i.e. node $A$ is $[A, A]$. This description is particularly useful, if node $A$ contains some hidden structure that is not seen by the application at this granularity level but is instead abstracted to a few aggregate measurements on $A$.

A path can be opened in only one of its side nodes. For instance, in Figure 1, path $[D, E, G)$ indicates a path that describes movement of articles from the time they enter location $D$ up to the point they enter hub $G$ (via $E$). Finally, $[A, G]^*$ is used as a shortcut for referring to the set of paths starting from node $A$ and ending in node $G$. This set is called a composite path. The notation is similar for open-ended paths. In what follows, we often omit the asterisk in the notation of a composite path, when there is no ambiguity in the context.

A maximal path in $G_q$ is defined as a path that is not contained in another path of $G_q$. Thus, graph query $G_q$ can be described as a set of maximal paths from the source nodes of $G_q$ to its terminal nodes. Let $Src(G_q)$ denote the set of source nodes and $Ter(G_q)$ the set of terminal nodes of graph $G_q$, respectively. The set of maximal paths of $G_q$ is $[Src(G_q), Ter(G_q)]^*$.

If the graph of Figure 1 is used to represent a query graph, then the source nodes are $\{A, B, C\}$, while the terminal nodes are $\{I, K\}$. For convenience, we often want to coalesce multiple nodes into a virtual aggregate node. As an example, let $R_2$ denote the subgraph of region 2 in Figure 1. Then, $[A, R_2)^*$ is used as a shortcut for the set of all paths from node $A$ to nodes in $R_2$, specifically $[A, D)$ and $[A, B, F)$.

In order to allow composition of paths, [10] introduces 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 $[A, B, F) \bowtie [F, J, K)$=$[A, B, F, J, K]$. On the contrary, path $[A, D, E]$ does not "join" with $[E, G, I]$ since they both include node $E$ and the resulting composition is not a path (the internal edge $[E, E]$ would be repeated otherwise). The operator is applied to composite paths as well by considering path-joins between all pairs of paths in them.

For instance, if we are only interested in articles that pass through all hubs of region 2 we can indicate all relevant paths using expression ($R_2$ is a graph denoting the nodes and edges in region 2)

$$[Src(G_q), Src(R_2)) \bowtie [Src(R_2), Ter(R_2)] \bowtie (Ter(R_2), Ter(G_q)]$$

This expression does not include path $[C, H, K]$ as the latter does not contain any location in $R_2$. Moreover, this expression permits us to reuse precomputed data on region 2, via the use of a materialized view, as will be explained in Section 5.

## 3.4 Path Aggregation

Analytical queries often need to perform ad-hoc aggregation of measures along existing paths they retrieve. Assuming that a user-defined function $F$ is given (for instance SUM() or MAX()) we define a path aggregation query $F_{G_q}$ as a shortcut for a process that retrieves all matching graph records for graph query $G_q$ and then applies function $F$ on all paths from a source node $s$ in $Src(G_q)$ to a sink (terminal) node $t$ in $Ter(G_q)$.

As an example, consider the three graph records depicted in Figure 2. We assume that the graph query $G_q$ is path $(A, C, E, F)$ and the SUM() function is used for path aggregation. Then, query $\text{SUM}_{(A,C,E,F)}$ retrieves record 2, as this is the only record that contains this path. Thus, for record 2, the result to this query contains path $(A, C, E, F)$ and the aggregate value of 7, which is computed by summing up the measures along the path.

An analytical query can use the result of a path aggregation and further consolidate the computed aggregates in order to compute higher level statistics, such as the average delivery time and the standard deviation for the retrieved records based on the order type, etc. Since this process is performed on the "flat" data returned from the underlying graphs, we consider it orthogonal to our techniques and can be easily handled in a relational store. *This is a distinct advantage of our methods, as the whole analysis can be performed in-house, within the relational realm.*

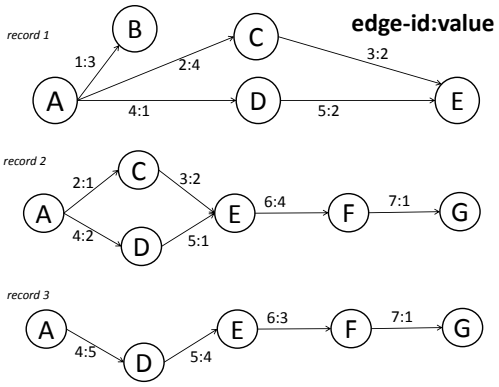



**Figure 2: Three sample graph records. The id and the associated measure is depicted over each edge**

# 4. GRAPH DATA IN A COLUMN-STORE

## 4.1 A Simple Storage Abstraction for Graph Records

It is well documented that traditional row-oriented relational stores are not suited for managing graph datasets, since navigation on the graph records requires successive join operators or recursion [9, 11]. Relational databases that utilize column-oriented storage of their tables alleviate these problems and permit us to model the graph records via a very simple relational schema. In particular, we utilize a single relational table $R(rec_{id}, m_1, m_2 \ldots, m_n)$ for storing all graph records. Attribute $rec_{id}$ is the key to this relation and uniquely identifies each record. Each attribute $m_i$ corresponds to the measure associated with the edge or node with identifier $i$. As explained, we assume that nodes and edges are identified with a unique numbering scheme. This information may be part of the metadata kept in the data store. Moreover, when no measure data is recorded by the application on certain elements (nodes or edges), the corresponding columns may be dropped from the schema. If a node or edge is not part of the particular record, this is indicated by a NULL value on the corresponding column.

Edges and nodes of a graph are treated identically in our framework, since a node $X$ can be considered as a special edge $[X, X]$. In what follows we will be referring to the structural elements of a record (nodes and edges) as "edges", since there is no distinction between them in our storage model.

Table 1 provides an example for the representation of the three records depicted in Figure 2. In the Figure, for each edge we depict its edge-id and the measure recorded on the respective graph record. In the relation, column $m_i$ is used for storing the measures of edge $e_i$. Columns $b_i$ are the respective bitmaps. The last three columns of the relation are associated with views. The details for these columns will be described in Section 5.

We note that, unlike applications in social networking or the semantic web, where graphs of massive scale need to be depicted, in our considered applications (CRM, WMS, SCM, etc) the number of distinct nodes and edges of the graph records is typically up to the order of a few thousands. Existing column-oriented RDBMS easily handle tables with such number of columns. If needed (see Section 6), this relation can be easily partitioned vertically (using $rec_{id}$s to link the partitions) in order to accommodate even larger number of distinct edges. Moreover, vertical compression of columns with many NULL values results in a small footprint for this particular storage abstraction, as will be demonstrated in our experiments.

| | ← measures → | | | | | | | ← bitmaps → | | | | | | | ← views → | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $r_{id}$ | $m_1$ | $m_2$ | $m_3$ | $m_4$ | $m_5$ | $m_6$ | $m_7$ | $b_1$ | $b_2$ | $b_3$ | $b_4$ | $b_5$ | $b_6$ | $b_7$ | $b_{v_1}$ | $m_{p_1}$ | $b_{p_1}$ |
| $r_1$ | 3 | 4 | 2 | 1 | 2 | NULL | NULL | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | NULL | 0 |
| $r_2$ | NULL | 1 | 2 | 2 | 1 | 4 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 5 | 1 |
| $r_3$ | NULL | NULL | NULL | 5 | 4 | 3 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 4 | 1 |

**Table 1: Content of master relation for the graphs of Figure 2**

## 4.2 Bitmap Columns for Efficient Retrieval of Graph Records

Retrieval of records for a given graph query $G_q(V_q, E_q)$ requires an efficient mechanism for locating records that contain the set of edges in $E_q$. Given the flat organization of the graph records we employ, bitmap indexes can be used for efficient query retrieval. In particular, for each edge $e_i$ a bitmap $b_i$ can be used to indicate records that contain the edge in consideration. Then, evaluation of query $G_q$ can be accommodated easily by ANDing the bitmaps of the edges in $E_q$. Thus, bitmap $b_q = AND_{e_{q_i} \in E_q} b_{q_i}$ computes the location of the graph records that are part of the result set of $G_q$.

The particular implementation of the bitmap indexes, depending on the particular data store used is orthogonal to our framework. In our implementation, we extend the schema of $R$ with $n$ columns of Boolean domain resulting in the following schema

$$R(rec_{id}, m_1, m_2, \ldots, m_n, b_1, b_2, \ldots, b_n)$$

Given the addition of these bitmaps, retrieval of the records that match query $G_q$ is done via an SQL statement of the form ($e_{q_i} \in E_q$ and $m_k$ is the measure associated with edge $e_k$):

SELECT $rec_{id}, m_{q_1}, \ldots, m_{q_m}$
FROM $R$
WHERE $b_{q_1} = 1$ AND ... $b_{q_m} = 1$

As it is evident, there is no need for a join in order to answer the query. More precisely, the requirement to "join" the constituent edges of the query graph is passed to the column-store engine, which is built around the idea of quickly joining columns of a vertically partitioned relation.

# 5. GRAPH VIEW MATERIALIZATION

## 5.1 Preliminaries

Evaluation of a graph query $G_q$ or a path aggregation query $F_{G_q}$ using the available bitmap indexes involves the retrieval of the bitmap columns $b_i$ from the master relation that correspond to the edges of the query graph in order to compute their intersection and locate the matching graph records. Even though each binary column $b_i$ is expected to easily fit in memory (the number of bits in a naive uncompressed representation of $b_i$ are equal to the number of graph records in the database), still the evaluation of a large query graph requires ANDing all binary vectors for that graph. Thus, the I/O cost attributed to these bitmaps for processing a graph query increases linearly with the number of bitmaps.

To improve response times, most data warehouses maintain the results of frequently needed queries in addition to the base records. These results are often referred to as *materialized views* [6, 12, 13] and are chosen by a view selection algorithm. This algorithm typically selects a subset of all possible views until a certain budget (such as the available disk space) is exhausted. In our work, we propose a *graph-view* selection framework that can be used to expedite processing of graph queries.

We will consider two variations of the problem depending on the type of queries. In the first, the workload consists of a set of graph queries $\mathcal{G}_q$=$\{G_{q_i}\}$. In the second variation of the problem, the workload consists of path aggregation queries $F_{\mathcal{G}_q}$=$\{F_{G_{q_i}}\}$. For both instances of the problem, materialized graph views are considered in order to expedite processing of the selected queries. The details of these views however vary depending on the instance.

While the techniques we present next are motivated by the use of views in data warehouses, the details of these views, their selection process and re-use in graph queries differ from existing works on materialized views, as will be explained.

### 5.1.1 Materialized Views for Graph Queries

Recall that evaluation of a graph query $G_{q_i}$ involves the retrieval of a number of bitmaps. Let $B_{G_{q_i}}$ denote the set of all these bitmaps and $B$ any subset of it ($B \subseteq B_{G_{q_i}}$). Let $bitmap(B)$ denote the bitmap resulting from ANDing all bitmaps $b_l \in B$. Then, as has been explained, evaluation of $G_{q_i}$ requires computation of $bitmap(B_{G_{q_i}})$. Using subset $B$ this can be expressed as

$$bitmap(B_{G_{q_i}}) = bitmap(B) \text{ AND } bitmap(B_{G_{q_i}} - B)$$

This formula simply suggest that we can precompute the conjunction of any set of bitmaps $B$ required for evaluating $G_{q_i}$ and store the result as a new bitmap. Then, we can reformulate the query so as to retrieve the graph records containing the query graph using a conjunction of the stored bitmap and the remaining bitmaps not considered in set $B$. $Bitmap(B)$ is called a graph view in our framework and can be added in the database schema as a new column in the master relation $R$. In SQL, we can treat $bitmap(B)$ as a regular bitmap column for a virtual edge associated with $B$ and reformulate the query accordingly. As an example, the subgraph of region 2 in Figure 1 can be indexed using a single bitmap column, which is computed by ANDing the bitmaps of the edges that are internal in region 2.

The benefit of using the graph view resulting from $B$ is that the number of bitmaps that need to be retrieved for evaluating the query is reduced by $|B| - 1$, $|B|$ being the number of bitmaps in set $B$. This is because a single bitmap (the graph view) is used instead of all bitmaps in set $B$. Taking this idea to the extreme, we can precompute a single bitmap $bitmap(G_{q_i})$ for each query in the workload. This solution will minimize the cost of processing these queries, however, it is not practical as the number of interesting graph queries can be very large. Moreover, many of these queries may share subgraphs, which means that excessive materialization may not only be impractical but also unnecessary. Thus, we need techniques that will select the "best" subset of graph views for the targeted application.

### 5.1.2 Materialized Aggregate Graph Views for Path Aggregate Queries

Path aggregate queries include a query graph $G_q$ and an aggregation function that is used in order to consolidate measures along paths in $[Src(G_q), Ter(G_q)]$. A materialized graph view for $G_q$ can expedite this query as well. Still, there is an opportunity for larger gains if, in addition to the bitmap of the view, the database also stores these pre-computed aggregates.

Let $p = \{e_1, e_2, \ldots, e_k\}$ be a path in $G_q$. This path may be

a maximal path in $[Src(G_q), Ter(G_q)]$ or any path within graph $G_q$. The resulting aggregate of the application of function $F$ in $p$ denoted as $F_p = F(m_1, m_2, \ldots, m_k)$ is a single value for each graph record containing $p$. For *algebraic* aggregate functions (such as the average) one can instead store the constituent distributive sub-aggregates (sum and count for the case of the average function) so that these pre-computations can be utilized in answering queries that are supergraphs of $G_q$.

Each aggregate value may be stored in the database, as a single column $m_p$ by extending the schema of the master relation $R$. Thus, a materialized *aggregate graph view* for a path aggregation query $F_{G_q}$ contains

- One column $m_p$. The content of this column is equal to $F_p$, for each record that contains $p$, or NULL otherwise.
- One binary column $b_p$ that instantiates a bitmap index for all graph records that contain $p$.

### 5.1.3 Summary

In what follows we will describe techniques for extending the schema of the database with materialized graph views. The resulting schema will be

$$R(rec_{id}, m_1, \ldots m_n, b_1, \ldots b_n, \underbrace{b_{v_1}, \ldots, b_{b_{v_k}}}_{\text{graph views}}, \underbrace{m_{p_1}, \ldots, m_{p_l}, b'_{p_1}, \ldots, b'_{p_l}}_{\text{graph aggregate views}})$$

Continuing the example of Table 1, there are two views materialized in the master relation. The first (column $b_{v_1}$) is a graph view corresponding to the subgraph containing edges $e_1, \ldots, e_4$. The second is an aggregate view for the path $p_1$=[$e_6$,$e_7$] and for the $SUM()$ function. The aggregated values on that path that is contained in records $r_2$ and $r_3$ are stored in column $m_{p_1}$=$m_6$+$m_7$, while $b_{p_1}$ is the corresponding bitmap.

## 5.2 Selection of Graph Views

Recall that a certain graph view is a bitmap indicating the existence of a particular set of edges in the graph records. Given a set of frequent graph queries $\mathcal{G}_q$=$\{G_{q_i}\}$, a naive approach is to compute the union $G_{all}$ of all query graphs $G_{q_i}$ and consider as candidate views all possible subsets of the edges in $G_{all}$. Since the number of possible subgraphs is exponential in the size of $G_{all}$, this approach is not practical. What we will show is that this exhaustive enumeration of candidate views is not even necessary because of a monotonicity property among the views.

Consider the simplest case where a single graph query $G_q$ is given. It is easy to verify that if we were to materialize a single graph view $G_v$, then the optimal scenario is to materialize the whole query, i.e. $G_v = G_q$. This is because this view (i.e. the corresponding bitmap), filters out all records but exactly those that contain the query graph. Assume we instead materialize a subset view $G_{v'} \subseteq G_v$. Then, in order to answer the single query in the workload we would need to retrieve the bitmap of $G_{v'}$ as well as the bitmap columns for all edges in $G_q - G_{v'}$.

Retrieval of a bitmap column in a column-store involves fetching from disk the corresponding bits. Since all bitmaps have exactly the same length (i.e. the number of graph records in the dataset), we utilize a simple but reasonable cost model, where the cost of fetching the bitmaps for a query is proportional to the number of bitmaps used in the formulation of the query. For a query, we also ignore the cost of fetching the remaining columns ($rec_{id}$ and measures related to $G_q$), as this cost is constant and is not affected by the reformulation of the query in the presence of a graph view. As has been explained, using a graph view $G_v$ with $k$ edges results in reducing the number of retrieved bitmaps by $k$-1, when no other view exists for the same query.

This analysis suggests that a graph view $G_v$ is of no use, if there is another larger view $G_{v'}$ that can be used for all queries that $G_v$ is useful for. In that case we say that $G_{v'}$ supersedes $G_v$:

*Monotonicity Property (Graph Views)*: Graph View $G_{v'}$ supersedes graph view $G_v$ (denoted as $G_v \prec G_{v'}$) iff $G_v \subset G_{v'}$ and $\forall G_q$ : $G_v \subseteq G_q \Rightarrow G_{v'} \subseteq G_q$.

From this discussion, the following observations are made regarding the generation of a set of candidate graph views $\mathcal{C}_v$ for a given workload $\mathcal{G}_q$=$\{G_{q_i}\}$.

- Each query graph $G_{q_i}$ needs to be considered for materialization. Even if there is another query $G_{q_j}$: $G_{q_i} \subset G_{q_j}$, this does not imply that for the corresponding views $G_{q_i} \prec G_{q_j}$. This is easy to prove by contradiction. If we assume that $G_{q_i} \prec G_{q_j}$, then for query $G_{q_i}$ it holds that $G_{q_i} \subseteq G_{q_i}$ and (because of the monotonicity property) it should also hold that $G_{q_j} \subseteq G_{q_i}$. This contradicts our assumption that $G_{q_i} \subset G_{q_j}$. Thus, $\mathcal{C}_v$ contains all graphs in $\mathcal{G}_q$.
- Let $G_{v_{i,j}}$=$G_{q_i} \cap G_{q_j}$ indicate the non-empty common subgraph of two query graphs $G_{q_i}$ and $G_{q_j}$, where $G_{q_i} \neq G_{q_j}$. Notice that $G_{v_{i,j}}$ is not superseded by $G_{q_i}$ (because of query $G_{q_j}$) nor by $G_{q_j}$ (because of query $G_{q_i}$). Thus, all subgraphs that are the intersection of two query graphs need also to be added to $\mathcal{C}_v$.
- Via similar arguments we would need to include in the set of candidate views $\mathcal{C}_v$ the common subgraphs between three or more query graphs in the workload.[1]

Because of the way set $\mathcal{C}_v$ is constructed, it follows that a graph view $G_v \in \mathcal{C}_v$ is not superseded by any-other graph view in the set. Thus, the computed set of candidate views, contains no redundant views. The set $\mathcal{C}_v$ is also minimal in the sense that removing a graph view $G_v$ from that set leads to sub-optimal decisions as we can find one or more queries in the workload $\mathcal{G}_q$, whose cost can be reduced further by including $G_v$ back in the set of candidate views.

While computation of set $\mathcal{C}_v$ is tractable, unlike the naive method described earlier, there is still a potential problem that arises when there is a lot of overlap among the graph queries. In the extreme scenario where there is overlap between any subset of query graphs, the number of candidate views is exponential to the number of graph queries: $|\mathcal{C}_v|$=$O(2^{|\mathcal{G}_q|})$. Even though this exponential dependency is in the number of graph queries (and not in the number of their edges, as in the naive enumeration), it may still be problematic for certain applications. For such cases, we propose a workaround that reduces significantly the cost of generating the set of candidate graph views that will be input to the graph view selection algorithm discussed later. Key to our solution is a modeling of the candidate view generation process as a frequent itemset counting problem. Assume that each graph query is a "set" of "items", where items correspond to graph edges in our case. Given a minimum support $minSup \geq 1$ we can compute (e.g. via the a-priori algorithm [14]) all frequent itemsets (an itemset is a set of edges, i.e. a graph view) whose support is equal or exceeds threshold $minSup$. The value of support in this formulation of the problem reflects the number of graph queries that a graph view can be used for. Because of the monotonicity property, we need in a post-processing step to filter out views that are superseded by other views in the result of the frequent itemset calculation process.

From the set of candidate views generated by either of the methods described above, we will select a subset of views for materialization. In our exposition, we concentrate on the case when a

[1]One can expedite this process by iteratively adding the common subgraphs of the common subgraphs identified in the previous steps, as was suggested by an anonymous reviewer.

space budget of $k$ views is defined and will be used to control the amount of precomputation in the database.

We will address this problem via an equivalent formulation as an extended set cover problem. Recall that our data store already contains a bitmap for each edge in the universe of edges stored in the dataset. Let $\mathcal{E}$=$\{\{e_1\},\ldots,\{e_n\}\}$ denote the set of all sets containing exactly one edge. Similarly, each candidate view is a set containing 2 or more edges. Let $\mathcal{S}$=$\mathcal{E} \cup \mathcal{C}_v$ be the union of the two sets of sets. To complete the mapping to an extended set cover problem, each query $G_{q_i}$ is termed a "Universe" $\mathcal{U}_i$ i.e. a set of edges of the corresponding graph query.

**Extended Set Cover Problem (multiple universes):** Find the minimum number of sets in $\mathcal{S}$ that can be used to cover all sets $\mathcal{U}_i$.

Finding the solution to the extended set cover problem will naturally lead to a selection of views: those sets of $\mathcal{C}_v$ that are part of the answer. Since there is no known algorithm that can solve this problem in polynomial time, we will rely on a greedy algorithm that chooses sets from $\mathcal{S}$ based on the number of uncovered edges in all universes. The greedy algorithm terminates when all query edges are covered. In our instance of the problem, when an upper bound of $k$ views is defined, will simply terminate after $k$ steps, or when a set from $\mathcal{E}$ (i.e. an existing bitmap for a single edge) is selected, whichever comes first. Thus, the complexity of the greedy algorithm is $O(\sum |U_i| \times k)$, i.e. it is linear in its input size (the size of the query graphs $\sum |U_i|$) and the value of parameter $k$.

## 5.3 Answering Queries from Views

The same greedy algorithm can be used at query time in order to select how to best answer (rewrite) a given query via the available views. In this case we have an instance of the typical set cover problem with a single universe (the query) and set $\mathcal{S}$ contains the atomic edges and graph views that are materialized in the database. The bitmaps corresponding to the sets in $\mathcal{S}$ selected by the algorithm will be used for answering the query. The solution provided by the greedy algorithm is an $H(n)$-approximation of the optimal solution, where $H(n)$ is the n-th harmonic number, for $n$ being the number of edges in the query graph.

## 5.4 Selection of Aggregate Graph Views

Recall that an aggregate graph view $F_p$ includes (i) a column for the aggregate measure computed for function $F$ along path $p$ and (ii) the bitmap for $p$, which is the conjunction of the bitmaps of its constituent edges. Thus, as in the case of graph views, aggregate graph views have all the same space requirements (two columns in this case). They differ however in that, in addition to replacing multiple bitmap computations, they also reduce the number of measures retrieved, since the stored aggregate can be used instead of the corresponding measures for the edges of path $p$. Thus, the longer the path $p$, the more gains we obtain, as more measures from the master relation are replaced by a single column. We here adopt a simple cost model that measures the benefit of an aggregate graph view proportionally to the length of path $p$ that the view instantiates. More formally, the benefit $benefit(F_p, F_{G_q})$ of using an aggregate view for a given query $F_{G_q}$ is measured proportionally (any monotone function will do for the analysis that follows) to the reduction in the number of columns that need to be retrieved from the column-store, if this view were materialized. This simple model is justified, since the practice shows that all bitmap columns have the same cost of retrieval in the column-store.

As in the case of graph views, the biggest challenge is to be able to compute the set of candidate views that will be given as input to the greedy algorithm. Clearly, one can not list all possible paths within the given set of query graphs, as their number is exponential to the size of the queries. Again, we will reduce the number of the candidate aggregate views by exploiting the following monotonicity property:

*Monotonicity Property (Aggregate Graph Views):* For any path $p_q$ and for any two aggregate graph views $F_{p_1}$, $F_{p_2}$ such that $p_1 \subseteq p_2 \subseteq p_q$: $benefit(F_{p_1}, F_{p_q}) \leq benefit(F_{p_2}, F_{p_q})$.

Intuitively, the monotonicity property states that materialization of larger paths leads to greater gains. Let $p_q$ be a maximal path contained in query $F_{G_q}$. Obviously, $F_{p_q}$ should be considered for materialization, as it is expected to help reduce the cost of the query. Let $p_{q'}$ be a sub-path of $p_q$ that does not overlap with any-other maximal path of the query. The monotonicity property asserts that the reduction of the query cost induced by $F_{p_{q'}}$ can not exceed the reduction obtained by using $F_{p_q}$. Thus, $F_{p_{q'}}$ should not be a candidate for materialization. On the other hand, if path $p_{common}$ is the intersection of two (or more) maximal paths then $F_{p_{common}}$ is a candidate aggregate view since its combined benefit for these paths may exceed the benefit from materializing one of the maximal paths.

Based on the above discussion, the set $\mathcal{C}_p$ of candidate paths, each corresponding to a candidate aggregate graph view is constructed as follows.

- Let $\mathcal{P}_{All}$ be the union of all maximal paths in the query workload. Let $\mathcal{G}_{All}$ be the graph containing all nodes and edges in the given set of queries. Notice that by definition repeated nodes and edges are ignored, thus $\mathcal{G}_{All}$ is not a multigraph.
- A node in $\mathcal{G}_{All}$ is *interesting* if it is the origin or endpoint of at least one maximal path in $\mathcal{P}_{All}$.
- A node is interesting if it is a starting node of two or more different edges traversed by two or more maximal paths in $\mathcal{P}_{All}$.
- A node is interesting if it is an ending node of two or more different edges traversed by two or more maximal paths in set $\mathcal{P}_{All}$.

The set of candidate paths $\mathcal{C}_p$ is constructed by considering all possible paths (of length greater than one) between interesting nodes in $\mathcal{G}_{All}$. Because of the monotonicity property, for any path $p$ (from a query graph) not included in this set, there is at least one candidate path $p_c$, whose corresponding view has benefit that is larger than that of path $p$ and can thus replace $p$ in a selection of materialized views.

As an example, in Figure 2 let's assume that the three graphs depicted are query graphs instead of graph records: $F_{G_{r_1}}$, $F_{G_{r_2}}$ and $F_{G_{r_3}}$, respectively, for some function $F()$. Then, the interesting nodes are $A$, $B$, $E$ and $G$. Consequently, the candidate paths are $[A,C,E]$, $[A,D,E]$, $[A,C,E,F,G]$, $[A,D,E,F,G]$ and $[E,F,G]$ resulting in 5 candidate aggregate graph views. Paths of length one (i.e. edges) like $(A,B)$ are not considered, since the database schema already stores their measures. A naive enumeration that considers all subpaths (of length greater than one) of the three query graphs would result in 11 candidate views instead of the 5 listed above.

Having the resulting set of candidate aggregate graph views and given a space budget of $k$ views, selection proceeds by using the same greedy algorithm described in the previous section.

# 6. DISCUSSION

## 6.1 Partitioning the Master Relation

In order to simplify the presentation, we have deliberately presented a simplified schema where all data, indexes and views are stored in a single relation, i.e. the "master relation" of Table 1. Of course, in practice storage of the graph records can be decoupled from the views and indexes by breaking the columns of this relation

into appropriate column families. Still, the single relation that will hold the graph records will contain as many columns as the number of distinct edges in the universe of the target application. Since these columns correspond to real world entities (hub locations in SCM, workflow states in WMS) we do expect that their numbers will be in the order of hundreds or thousands rather than millions as in a social networking application. In order to cope with domains where larger graph records may arise, in our implementation we utilize a simple vertical partitioning scheme, where the master relation is automatically broken into sub-relations with up to 1 thousand columns each. Intelligent clustering of these columns based on the users' query patterns is possible but the details are beyond the scope of this paper. Partitioning allows us to handle (in our experiments) graph records of up to 100K edges each, even-though such records may never appear in practice in the applications of interest. Moreover, it should be evident from the discussion that the schema we propose is not necessarily static but can be expanded on demand, in case new columns are required for the newly added records.

## 6.2 Managing Arbitrary Graphs

The presented storage model handles graphs, with no difference if there are cycles or not. The framework assigns a table column to each edge. If there are two directed edges $(A, B)$ and $(B, A)$ or, in general, back edges that form a cycle this makes no difference to the storage model. For instance, edges $(A, B)$ and $(B, A)$ are simply mapped to two different columns. Similarly, for graph queries (with no aggregation), the presence of cycles doesn't make any difference in their formulation and answering processes.

On the contrary, cycles matter when doing path aggregation. In that case our framework requires, in a pre-processing step, flattening of each record into a directed acyclic graph (DAG). As an example, consider a product, which is successively shipped to the locations $A$, $B$, $C$, $A$, $D$ and $E$, in that order. Removing the cycle results in a a second copy of node $A$ and the following sequence of edges $(A, B)$, $(B, C)$, $(C, A')$, $(A', D)$ and $(D, E)$. This is necessary when, for example, the aggregate function tries to compute the total time along paths that connect two nodes. In the previous example, it helps distinguish that we are computing the total time starting from the first occurrence of $A$ and not $A'$ (i.e. the first time the package left that location). Thus, flattening of the graph into a DAG is performed so that aggregation on sequences of nodes/edges works in a desirable manner in the applications of interest. In many applications (such as in SCM with RFID tags used in tracing the various products) sequencing of the nodes/edges is already encoded in the data, for instance using time-stamps recorded by the RFID readers.

## 6.3 Incorporating Specialized Graph-Indexes

In the literature there exists a substantial amount of related work on graph indexing targeting, in most cases, graph isomorphism queries (see discussion of Section 8). Our target applications do not require such query primitives and, moreover our aim is at analytical queries that aggregate the collected measures. Still, there is a simple way to extend our framework so as to incorporate many of the existing graph indexes. As an example, the gIndex [5] is using a set of features to index the graph records, by recording the graphs that contain each feature. In our representation we can utilize a bitmap column for each such feature using 1's to indicate those records that contain it. For the particular example of gIndex, it utilizes a set of features termed *discriminative fragments*. In our implementation, we can optionally utilize the gIndex in the form of additional bitmap columns (one per fragment) that can be populated when loading the records.

In our presentation we have instead opted for a very basic form of indexing based on atomic edges (the bitmap columns). This form of indexing is practicable in our considered applications where massive collections of small graph records need to be injected. Populating the bitmap columns depicted in Table 1 is computationally straightforward. Selection of the fragments on the other hand requires running an expensive graph mining algorithm [5], something that may not be feasible for the data sizes we consider in our work. Moreover, the simple bitmaps we utilize, as will be evident by our experiments, are very effective, even when querying hundreds of millions of graph records. These experiments also show that for graph queries and even more for those that involve aggregation, materialized views are more efficient than indexes, because they substantially reduce the number of measures that need to be retrieved from disk due to pre-aggregation.

| | NY | GNU |
|---|---|---|
| Number of graph records | 320 Million | 100 Million |
| Total number of measures | 27.3 Billion | 7.5 Billion |
| Size on disk | 241 GB | 68 GB |
| Distinct number of edge ids | 1000 | 1000 |
| | up to 100K in sensitivity tests | |
| Min. number of edges per record | 35 | 45 |
| Max. number of edges per record | 100 | 100 |
| | up to 10K in sensitivity tests | |
| Avg. number of edges per record | 85 | 75 |

**Table 2: Description of Datasets and Default Values**

# 7. EXPERIMENTS

## 7.1 Experimental Set Up

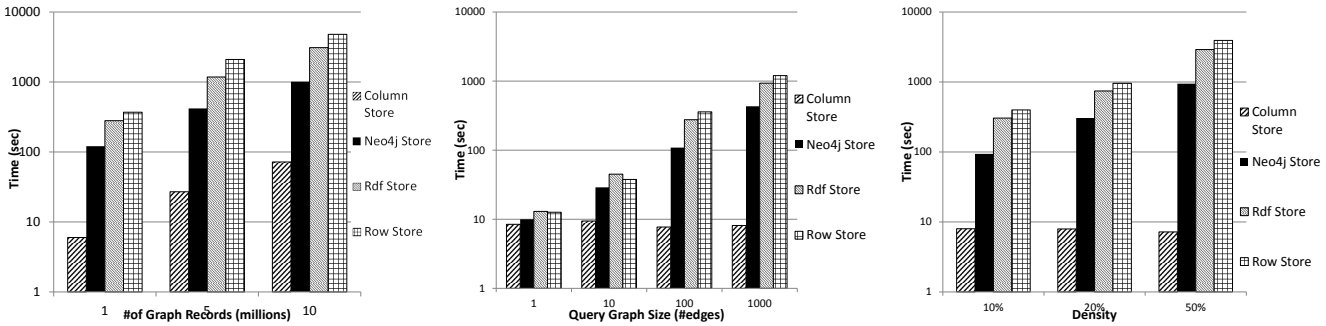
In this section we provide an experimental evaluation of the proposed framework. All experiments were executed using a PC with a single i7 860 (2.8GHz) processor, 8GB of memory and a single 1TB 7200rpm HDD. While the choice of column-store is orthogonal to our framework, for these experiments we downloaded and used the popular MonetDB [15] in its default configuration. We worked with the following two datasets.

- **NY:** This dataset depicts New York roads and was downloaded from: http://www.dis.uniroma1.it/∼challenge9/download.shtml.
- **GNU:** This dataset describes connections between Gnutella hosts and is available at: http://snap.stanford.edu/data/p2p-Gnutella04.html.

From each dataset we synthesized millions of graph records. The graph records were generated by invoking multiple random walk processes in the underlying graphs. We then assigned a random real value to each of their edges, to be used as a measure for aggregation. In Table 2 we provide several statistics on the graph records generated from the two datasets.

Our selection of datasets is justified as follows. The NY dataset may be used to describe a distribution network within the city in a SCM application. 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 or subnets.

In our experiments we used query graphs that are generated either with uniform or with Zipf distribution from the set of paths resulting from the random walk processes. We used the $SUM()$ function for path aggregation. All experiments were executed 5 times (starting each time with a "cold" system, after a reboot) and we report the average numbers.




(a) Dataset Size (b) Query Size (c) Density

**Figure 3: Query Execution Times in Alternative Systems**

## 7.2 Sensitivity Analysis, Comparison Against Alternative Implementations

In what follows we provide a sensitivity analysis of our techniques to various parameters. For comparison, we also include results for (i) a straightforward implementation that uses a commercial RDBMS and row-oriented storage for storing the graph records using triplets of record id, edge id and measure values and appropriate indexes, (ii) an implementation using the neo4j native graph database, and (iii) a commercial RDF data store. For each system we used the default/suggested configuration parameters given by its provider. As performance of the alternative systems was orders of magnitude slower than our implementation, we used smaller subsets of the NY dataset consisting of 1, 5 and 10 million graph records. In the next subsection, we discuss performance of our implementation for the full-scale datasets.

Figure 3(a) depicts the overall execution time for sets of 100 uniform graph queries. Performance of our system scales linearly with the size of the dataset, as for a fixed set of queries, the result set also scales linearly. In comparison, queries on the row-store are orders of magnitude slower. The non-relational systems performed better than the row store database but they were both significantly slower than our implementation.

In Figure 3(b) we show the query execution times for 100 queries, when we vary the number of edges in the query graphs from 1 to 1000 for the dataset with 1 million graph records. Unlike the other implementations, performance of the column-store becomes even better with increasing the query graph sizes. This is explained as fewer records are retrieved in response to a more complex graph query. Thus, the overhead of retrieving more bitmap columns for evaluating the structural condition is offset by the reduced I/O for retrieving the corresponding measures.

We next study the effect of data graph record density on the four storage platforms. We used subsets from the NY data consisting of 1 million graph records where the number of distinct edge-ids is 1000. We varied the density of the graph records, where the latter is defined as the percentage of edges used in a record. For instance, a density of 20% means that a graph record contains 200 edges. Query graphs are also constructed for varying density factors. In Figure 3(c) we can see that increasing density does not affect the column-store because of a similar increase in the size of the query graphs (see also Figure 3(b)). As expected, the size of the database (Figure 4) increases linearly for the row-oriented data store. In the column-store, the database size is independent on the graph record density and remains constant. In comparison, the native graph database (Neo4j) seems to require the most disk space for storing these records.

In Figure 5 we increase the size of the universe of edges from 1K up to 100K. Each dataset contained 10M graph records with 10% density. This implies that graph records are getting larger with increasing the domain of edges, and query output size increases proportionally. As discussed in Section 6, in our implementation we automatically break the master relation into sub-relations when the number of its columns exceeds 1000. In that case, retrieval of the graph records requires joining these sub-relations. For the rightmost data point in the Figure there are 100 sub-relations to join. This is the reason why performance of the column store is getting worse, by increasing the number of distinct edges. For comparison we depict the query performance of neo4j, in which query run time increases linearly, mainly because of the increase in query output. This Figure suggests that the column-store performs better even for domains of up to 100K distinct edges, eventhough such large domains are not typical in the applications of interest. As will be shown next, by the use of materialized graph views, its query performance becomes even better.

## 7.3 Benefits of Graph Views

We now examine the impact that the materialization of graphs views has on query answering. These experiments were run using the full datasets described in Table 2. We used random sets of 100 queries each and measured their wall-clock execution time increasing the number of graph views materialized. The results for uniform queries are shown in Figure 6 for the NY dataset. The x-axis depicts the "space budget" and represents the number of views (bitmap columns) added as a percentage of the queries. The rightmost entry (100%) corresponds to the case where 100 bitmaps are added. In MonetDB, a bitmap column (a single view) is roughly 0.02% of the database size, when no views are present. This means that in the extreme case, when all 100 views are materialized, we expand the database size by only 2%. The y-axis depicts the cumulative execution time for all 100 queries. The graph for the GNU dataset is similar (queries are faster though because of the smaller dataset size) and is omitted due to lack of space.

In the graph we break down the total query execution times in two parts. The bottom part in the charts involves the retrieval of the measures requested by the queries. A set of 100 queries returns on the average 30 Million graph records from the NY dataset and 11 Million graph records from the GNU dataset. These graph records contain about 2.5 Billion and 0.8 Billion measures respectively. Since, no aggregation is performed by these graph queries, the cost of retrieving these measures is mandatory and is not affected by the use of graph views. The views merely act as indexes for these queries, affecting the remainder upper part of the execution times break-down that is shown in the Figure. Considering the

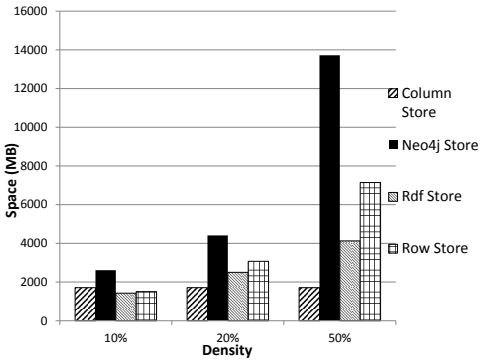



**Figure 4: Disk Space vs Density**

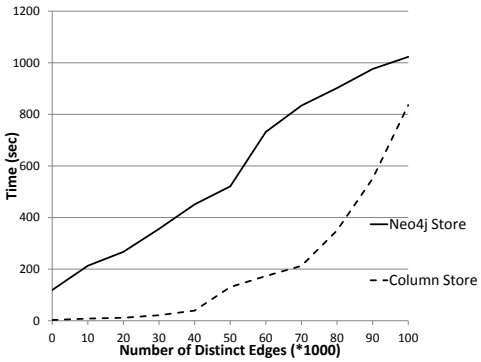



**Figure 5: Query Times vs Edge Domain Size**

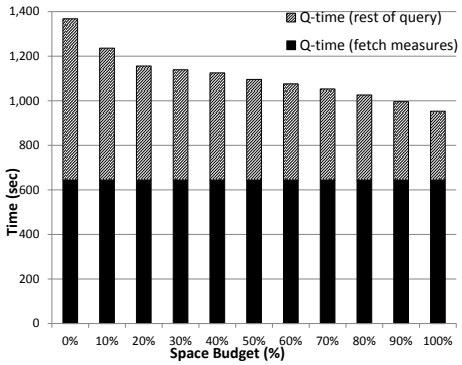



**Figure 6: Run Time (100 Uniform Graph Queries) vs Space Budget, NY Dataset**

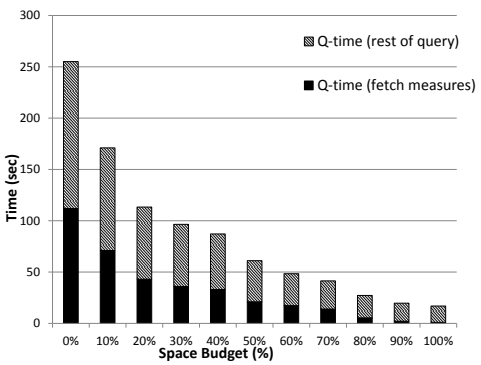



**Figure 7: Run Time (100 Uniform Aggregate Graph Queries) vs Space Budget, GNU Dataset**

massive size of the results of these queries, execution times even without any view materialized are small for a single mechanical drive and with no clustering for these records: each query takes about 13.5 seconds on the NY dataset and about 4.7 seconds on the smaller GNU dataset. This performance is explained by the use of the bitmap columns our framework utilizes for indexing the edges of each graph record. Still, the use of graph views provides significant benefits in reducing the execution times further, by up to 32%. If we take out of the equation the mandatory cost of fetching the results, the reduction is up to 57%.

In Figure 7 we repeat the experiment, this time using queries that perform path aggregation on the returned graphs on the GNU dataset. Since an aggregate graph view consolidates all measures along the edges of the path into a single value, using the views results in fewer values (original measurements or aggregated values) retrieved by the queries. This is indicated in the results where we, once again, break down the running time of the queries depicting the time spent on retrieving the measures. As expected, the benefits of using the views are more profound in that case. For instance using 100 views results in reducing the execution time of the query workload by up to 89%. For this data, the size of an aggregate graph view is about 0.1% of the database size. Thus, the full space budget for the views corresponds to an increase of the database size by about 10%.

In Figure 8 we repeat the experiment using zipf-distributed queries. In the graph we plot the relative execution times for 100 queries (simple or aggregate) for the two datasets. The relative time is computed as the fraction of the total time for obtaining the results over the execution time for the same queries without the use of views. When queries exhibit skew, there is increased sharing among the graph queries resulting in larger gains when using materialized graph views for the same space budget. Overall, there is a reduction in the execution times of up to about 34% for simple (non-aggregate) queries and up to 94% for aggregate queries.

In Figure 9 we report the total number of candidates generated for the NY dataset and for different query types and distributions, when we vary the minimum support $minSup$ of a candidate view as explained in Section 5.2. Computation of the candidate views in our datasets took less than one second, independently of the value of $minSup$ used. We do not present results for a naive enumeration of the candidates, as this is not computationally feasible for this workload. We notice that an initial increase of the value for $minSup$ sharply reduces the number of candidates.

In Figures 10, 11 we extend our framework to include additional bitmap columns, selected by the gIndex technique, as explained in Section 6.3. We used gSpan [16] for mining frequent subgraphs and then we selected from them the set of discriminative fragments according to the default parameters presented in [5]. Since mining of frequent graphs is a lengthy process that cannot be completed on a dataset containing hundreds of millions of graphs, we were forced to utilize a small subset of the NY dataset containing 10M records, using a sample of 1% for mining. For this sample size (100K graph records), identification and selection of the fragments took 1.5 hours on our machine. In comparison, our view selection algorithm ran in under a second. Because of the small size of the sample, in order to further increase the benefits of these fragments we utilized two different processes for selecting the sample. For the line depicted as $gIndex_Q$ in the Figures, we trained the index using only records that are part of the query results (for the respective queries used in each experiment), seeking to obtain a selection

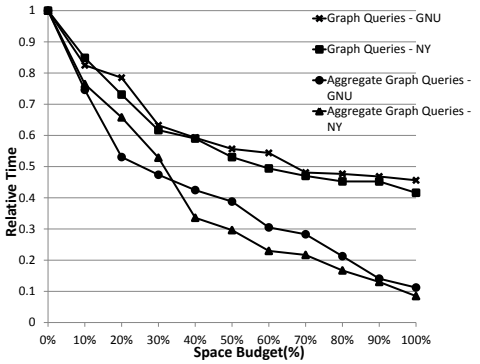



**Figure 8: Relative Performance of Zipf Queries**

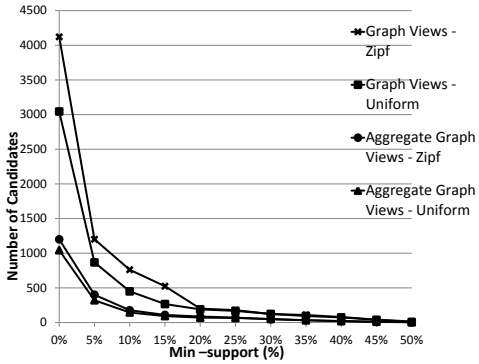



**Figure 9: Number of Candidate Views, NY dataset**

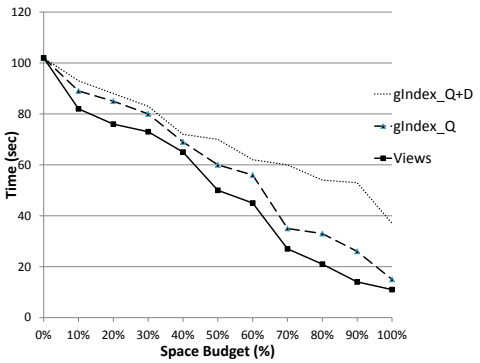



**Figure 10: Use of gIndex, 100 Uniform Graph Queries**

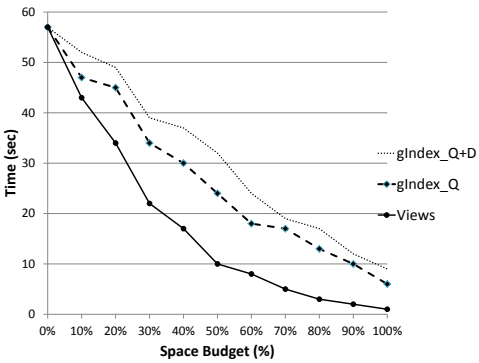



**Figure 11: Use of gIndex, 100 Uniform Aggregate Graph Queries**

of gIndex fragments tailored for these queries. Alternatively, we trained the index using 80% of random records and 20% of records that answer the queries (line $gIndex_{Q+D}$). In the Figures we vary the number of discriminative fragments (columns) selected. All setups include the use of the original bitmap columns on the edges that we promote in our framework, since without them query execution often results in a full scan of the database. The Figures suggest that using the fragments selected by the gIndex technique improves reducing query execution times, however this reduction cannot reach the benefits provided by using the same number of views, especially for analytical queries that perform aggregation. On these queries use of the views results in up to 6 times faster query performance compared to using $gIndex_Q$. We also tested combining both gIndex and the views on the same query, which for some queries resulted in marginally smaller execution times than using only the views.

# 8. RELATED WORK

In the database literature there is significant work on recursive queries (e.g. [11, 17, 18]), which have been our motivation. The work of [9] first discussed measure aggregation over simple path expressions in a graph. The queries we consider in our framework are more complex as (being themselves graphs) they can define arbitrary structural conditions for the retrieved records.

Graph summarization operators are discussed in [9, 19, 20]. Such techniques are complementary and they may be used in conjunction to our framework. In the recent work of [21], the authors develop a novel graph OLAP framework that assumes linked set of tuples that are described via a graph model (e.g. the authors in DBLP, where links may describe collaboration frequencies). In contrast, we do not assume the existence of a single graph, but we rather look at applications that generate millions of graph records. Still, these techniques may be used with our framework, for example in order to derive some average graph over a set of customers.

Recent work on large-scale graph processing (e.g. [22, 23, 24]) focuses on applications where a single massive graph structure is processed. These techniques rely on available parallelism in order to perform bulk computations on this graph. In contrast, our applications of interest generate massive collections of independent graph records. Efficient ad-hoc query processing on these records demands proper indexing and pre-computation techniques that are both provided via the graph view framework we introduce.

Graph data can also be depicted using the RDF data model, for instance treating each edge as a separate triplet. Implementation-wise, there are two main classes of RDF triplet stores. The first includes systems that natively store the RDF data using proprietary binary formats (e.g. Jena TDB, RDF-3X, 4store) and the second class involves implementations over a relational backend. Systems of the second class try to leverage proven features of the relational systems such as scalability, transaction support, security, etc that are often lacking in native stores. One of the most important questions for relationally-backed stores is how to shred the RDF schema into relational tables. A first alternative is to use a single massive table for storing the triplets. Despite the simplicity of this approach, querying requires multiple joins, just as is the case of querying a graph database stored in a single edge-adjacency relation. A second alternative is to create multiple relations per RDF data type [25] or predicate [26]. Recently, a hybrid approach has been proposed [27] that clusters multiple predicates into the same column of a relation. While our techniques follow a similar idea of shredding the graphs into a relational store, the particular details of this process are different. Moreover, the aforementioned techniques assume the exis-

tence of an RDF schema in order to guide the shredding, while in our applications, we do not assume prior knowledge of the process that generates the graph records. Finally, the type of queries we address in this work are tailored to the needs of the applications that generate the graph records and not generic SPARQL statements.

In the literature, there are many works in graph indexing so as to support subgraph isomorphism testing. In [5] an index called gIndex makes use of frequent substructures as the basic indexing feature. Sapper [28] proposed a subgraph indexing and matching method to find the occurrences of a query graph in a large database graph with possible missing edges. TreeSpan [29] conducts similarity all-matching using a minimal set of spanning trees for query $q$ to cover all connected subgraphs of $q$ missing at most $k$ edges.

The aforementioned works differ from our framework in several aspects. First, they consider the problem of graph isomorphism, which is important for applications such as computational biology but is not central in our setting. In our work there is no need to match edges and paths between queries and the graph database as they are known a priori. Moreover our framework retains the benefits of having the data in a relational repository, enabling better integration of the analysis of these records with additional data already stored in the database. Our techniques are suitable for massive disk resident datasets. In contrast, many of the aforementioned techniques do not directly discuss storage and retrieval of the graphs (or indexes they built) from secondary storage. In our techniques we utilize existing relational technology enabling processing of graph data that is orders of magnitude larger than the available memory. Graph indexing techniques require substantial preprocessing of the data in order to mine frequent features such as subgraphs, trees or paths. This process may be prohibitively time consuming in large datasets (this limitation is evident in the sizes of the graph databases used in these works) in order to build the index. In comparison our techniques build simple indexes in the form of bitmap columns and materialized views that are all computed in a single pass.

# 9. CONCLUSIONS

Graph data is becoming popular due to emerging applications that need to store, process and manipulate complex datasets. In this paper we proposed an intuitive framework where both data and queries are modeled as abstract graph structures. We argued for and demonstrated in our experiments that relational systems are efficient in storing graph records given an intuitive flat representation of these graphs backed by a a column-store and paired with novel use of bitmap indexes and materialized graph views we introduced. Our experiments using very large graph databases showed that our platform is orders of magnitude faster not only compared to a straightforward relational implementation but also than alternative systems that natively handle graph data.

# 10. REFERENCES


[1] B. Amann and M. Scholl, "Gram: A Graph Data Model and Query Languages," in *Proceedings of the ACM conference on Hypertext*, New York, NY, USA, 1992, pp. 201–211.

[2] R. H. Güting, "Graphdb: Modeling and Querying Graphs in Databases," in *VLDB*, 1994, pp. 297–308.

[3] D. Bleco and Y. Kotidis, "RFID Data Aggregation," in *GeoSensor Networks*, 2009, pp. 87–101.

[4] P. E. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes," in *SIGMOD Conference*, 1997.

[5] X. Yan, P. S. Yu, and J. Han, "Graph Indexing: A Frequent Structure-based Approach," in *Proc. of SIGMOD*, 2004.

[6] Y. Kotidis and N. Roussopoulos, "A Case for Dynamic View Management," *ACM Trans. on Database Systems*, 2001.

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

[8] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu, "View Selection in Semantic Web Databases," *PVLDB*, vol. 5, no. 2, pp. 97–108, 2011.

[9] Y. Kotidis, "Extending the Data Warehouse for Service Provisioning Data," *Data & Knowledge. Engineering*, vol. 59, no. 3, pp. 700–724, 2006.

[10] D. Bleco and Y. Kotidis, "Business Intelligence on Complex Graph Data," in *BEWEB*, 2012, pp. 13–20.

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

[12] N. Roussopoulos, "Materialized Views and Data Warehouses," *SIGMOD Record*, vol. 27, no. 1, 1998.

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

[14] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," in *VLDB*, 1994.

[15] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012.

[16] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining," in *ICDM*, 2002, pp. 721–724.

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

[18] 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.

[19] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient Aggregation for Graph Summarization," in *SIGMOD*, 2008.

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

[21] 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.

[22] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: Mining Peta-scale Graphs," *Knowl. Inf. Syst.*, 2011.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed Graphlab: A Framework for Machine Learning in the Cloud," *PVLDB*, 2012.

[24] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *ACM SIGMOD*, 2010.

[25] K. Wilkinson, C. Sayers, H. A. Kuno, and D. Reynolds, "Efficient RDF storage and retrieval in Jena2," in *SWDB*, 2003.

[26] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach, "Scalable semantic web data management using vertical partitioning," in *VLDB*, 2007, pp. 411–422.

[27] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient RDF store over a relational database," in *SIGMOD Conference*, 2013, pp. 121–132.

[28] S. Zhang, J. Yang, and W. Jin, "SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs," in *VLDB*, 2010.

[29] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu, "TreeSpan: Efficiently Computing Similarity All- Matching," in *SIGMOD Conference*, 2012, pp. 529–540.