Online Analytical Processing on Graphs (GOLAP): Model and Query Language

Seyed-Mehdi-Reza Beheshti[1]
Boualem Benatallah[1]
Hamid Reza Motahari-Nezhad [2]
Mohammad Allahbakhsh [1]

[1] University of New South Wales
Sydney 2052, Australia
{sbeheshti,boualem,mallahbakhsh}@cse.unsw.edu.au

[2] HP Labs Palo Alto
CA 94304, USA
hamid.motahari@hp.com

Technical Report
UNSW-CSE-TR-201214
May 2012

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Graphs are essential modeling and analytical objects for representing information networks. Examples of this are the case of Web, social, and crowdsourcing graphs which may involve millions of nodes and relationships. In recent years, a new stream of work has focused on On-Line Analytical Processing (OLAP) on graphs. Although these line of works took the first step to put graphs in a rigid multi-dimensional and multi-level framework, none of them provide a semantic-driven framework and a language to support n-dimensional computations on graphs. The major challenge here is how to extend decision support on multidimensional networks considering both data objects and the relationships among them. Moreover, one of the critical deficiencies of graph query languages, e.g. SPARQL, is lack of support for n-dimensional computations which are frequent in OLAP environments. Traditional OLAP technologies were conceived to support multidimensional analysis, however, they cannot recognize patterns among graph entities and analyzing multidimensional graph data (from multiple perspectives and granularities) may become complex and cumbersome. In this paper, we present a framework, simple abstractions and a language to apply OLAP style queries on graphs in an explorative manner and from various user perspectives. We redefine OLAP data elements (e.g., dimensions, measures, and cubes) by considering the relationships among graph entities as first class objects. We have implemented the approach on top of FPSPARQL, Folder-Path enabled extension of SPARQL. The evaluation shows the viability and efficiency of our approach.

1 Introduction

In traditional databases (e.g., relational DBs), data warehouses and OLAP (On-Line Analytical Processing) technologies [1, 13] were conceived to support decision making and multidimensional analysis within organizations. To achieve this, a plethora of OLAP algorithms and tools have been proposed for integrating data, extracting relevant knowledge, and fast analysis of shared business information from a multidimensional point of view. Moreover, several approaches have been presented to support the multidimensional design of a data warehouse. Cubes defined as set of partitions, organized to provide a multi-dimensional and multi-level view, where partitions considered as the unit of granularity. Dimensions defined as perspectives used for looking at the data within constructed partitions. Furthermore, OLAP operations have been presented for describing computations on cells, i.e. data rows. Unlike traditional data warehouses, where the focus is on storage and retrieval of data items, graph databases used to represent information-rich, inter-related and multi-typed networks that support comprehensive data analysis.

In recent years, a new stream of work [28, 29, 52, 16, 58, 43, 34, 33, 21] has focused on on-line analytical processing on graphs. Although these line of works took the first step to put graphs in a rigid multi-dimensional and multi-level framework, none of them provide a semantic-driven framework and a language to support n-dimensional computations on graphs. The major challenges here are: (i) how to extend decision support on multidimensional networks considering both data objects and the relationships among them: traditional OLAP technologies cannot recognize patterns among graph entities and, consequently, enabling users to analyze multidimensional graph data may become complex and cumbersome. To address this challenge we present a framework and simple abstractions for modeling on-line analytical processing on graphs. We consider the relationships among graph entities as first class objects; and (ii) providing multiple views at different granularities is subjective: depends on the perspective of OLAP analysts how to partition graphs and apply further operations on top of them. To address this challenge we extend SPARQL [41] (the official W3C standard query language for RDF graphs) to support n-dimensional computations on graphs.

The unique contributions of the paper are as follows:

- We propose a graph data model, *GOLAP*, for online analytical processing on graphs. This data model enables extending decision support on multidimensional networks considering both data objects and the relationships among them. We use the notions of folder and path nodes (previous work of authors [10]) to support multi-dimensional and multi-level views over large graphs. We redefine OLAP data elements (e.g., dimensions, measures, and cubes) by considering the relationships among graph entities as first class objects.

- We extend SPARQL to support n-dimensional computations on graphs. We introduce the new clause 'GOLAP', to query and analyze GOLAP graphs. This clause supports partitioning graphs (using folder and path nodes) and allows evaluation of OLAP operations on graphs independently for each partition, providing a natural parallelization of execution. We propose two types of OLAP operations: *assignments*, to apply operations on entity attributes, and *functions*, to apply operations on network structures among entities. GOLAP operations support UPDATE and UPSERT semantics. We describe optimizations and execution strategies possible with the proposed extensions. We provide a front-end tool for assisting users to create GOLAP queries in an easy way.

The remainder of this paper is organized as follows: We present background and related work in section 2. Section 3 presents an example scenario. In section 4 we present a graph data model for online analytical processing on graphs. In section 5 we propose a query language for applying OLAP operations on graphs. In section 6 we describe the query engine implementation, architecture and evaluation experiments. Finally, we conclude the paper with a prospect on future work in Section 7.

2 Background and Related Work

OLAP (On-Line Analytical Processing) [1, 13] is part of the broader category of business intelligence and were conceived to support information analysis using data warehouses in order to extract relevant knowledge of organizations. OLAP applications typically access large (traditional) databases using heavy-weight read-intensive queries. OLAP encompasses *data decision support* (focusing on interactively analyzing multidimensional data from multiple perspectives) and *data mining* (focusing on computational complexity problems). There have been a lot of works, discussed in a recent survey [44] and a book [51], dealing with multidimensional modeling methodologies for OLAP systems. Multidimensional conceptual views allow OLAP analysts to easily understand and analyze data in terms of facts (the subjects of analysis) and dimensions showing the different points of view where a subject can be analyzed from. These line of works, propose OLAP data elements such as partitions, dimensions, and measures and their classification, e.g. classifying OLAP measures into distributive, algebraic and holistic. They discuss that one fact and several dimensions to analyze it give rise to what is known as the data cube. There are many works, e.g. [27, 57, 55, 24], dealing with the efficient computation of OLAP data cubes. Many other works, e.g. [3, 37, 23], deal with clustering/partitioning large databases, as OLAP queries are typically heavy-weight and ad-hoc thus requiring high processing power. And many other works, e.g. [6, 54], focused on querying multidimensional models, to analyze independent data tuples that mathematically form a set, i.e. conventional spreadsheet data.

In recent years, a new stream of work [28, 29, 52, 16, 58, 43, 34, 33, 21] has focused on online analytical processing of information networks[1]. Chen and Qu et. al. [16, 43] proposed a conceptual framework for data cubes on graphs and classify their framework into informational (dimensions coming from node attributes) and topological (dimensions coming from node and edge attributes) OLAP. Although these works took the first step to put graphs in a rigid multi-dimensional and multi-level framework, none of them provide a semantic-driven framework and a language to support n-dimensional computations on multiple views and different granularities of graphs. Tian et. al. [52] proposed operations to produce a summary graph (by grouping nodes) and controlling the resolutions of summaries (by providing the drill-down and roll-up abilities). They did not show how the proposed framework can support OLAP analysts to define dimensions coming from the attributes of network structures among entities. Zhao et. al. [58] introduced a new data warehousing model, Graph-Cube, that supports OLAP queries on graphs. They considered both attribute aggregation and structure summarization of the networks. We use and extend proposed Graph-Cube to provide multiple views at different granularities by introducing simple abstractions and a language for the explorative querying of on-line analytical processing on graphs. Kämpgen et. al. [33] presented a mapping from statistical Linked Data that conforms to the RDF Data Cube vocabulary. The authors did not address how the mapping framework supports semantics for set of dimensions coming from the attributes of topological

[1] An information-network [28] is a network where each node represents an entity (which may have attributes, labels, and weights) and each link (which may have rich semantic information) represents a relationship between two entities.

elements, i.e., nodes and edges of the graph. Etcheverry et. al. [21] introduced Open Cubes, an RDFS vocabulary for the specification and publication of multidimensional cubes on the Semantic Web. The authors focused on informational attributes and did not show how the aggregation and OLAP operations can be performed along the topological dimensions defined upon the network.

Another line of related work [19, 22, 2, 41, 31, 5, 35, 10] focused on mining and querying information networks. Some of existing approach for querying and modeling graphs [19, 22] focused on defining constraints on nodes and edges simultaneously on the entire object of interest, not in an iterative one-node-at-a-time manner. Therefore, they do not support querying nodes at highler levels of abstraction. BiQL [19] focused on the uniform treatment of nodes and edges in the graph and supports queries that return subgraphs. SPARQL [41] is a declarative query language, an W3C standard, for querying and extracting information from directed labeled RDF[2] graphs. SPARQL supports queries consisting of triple patterns, conjunctions, disjunctions, and other optional patterns. However, there is no support for querying grouped entities. Paths are not first class objects in SPARQL [41, 31]. PSPARQL [5] extends SPARQL with regular expressions patterns allowing path queries. SPARQLeR [35] is an extension of SPARQL designed for finding semantic associations (and path patterns) in RDF bases. FPSPARQL [10], Folder-Path enabled extension of SPARQL, supports folder and path nodes as first class entities that can be defined at several levels of abstractions and queried.

Other works [48, 2, 42, 49, 50, 32] focused on clustering and classification of networks by studying systematically the methods for mining information networks. Graph clustering algorithms, e.g. [48, 2, 42], are of two types: node clustering, in which algorithms determine dense regions of the graph based on edge behavior, and structural clustering, in which algorithms attempt to cluster different graphs based on overall structural behavior. Also, there are some works in clustering, e.g. [49, 50], developed a ranking-based clustering approach that generates interesting results for both clustering and ranking techniques [28]. Classification techniques [2, 32] classify graphs into a certain number of categories by similarity. These line of works classify networks based on the fact that nodes that are close to similar objects via similar links are likely to be similar. All these works provide some kind of (network) summaries incorporates OLAP-style functionalities.

In this approach, we focus on providing users with an explorative method to analyze multidimensional graph data from multiple perspectives and granularities. We use folder and path nodes (our previous work [10]) to provide network summaries and to support multidimensional and multi-level views over graphs. We present a query language to query and analyze online analytical processing on graphs in an explorative manner.

3 Example Scenario

DBLP (Digital Bibliography and Library Project) is a computer science bibliography database. DBLP listed more than 1.8 million publications (in May 2012) and tracked most of journals and conference proceedings. We use DBLP[1] dataset to generate graph models containing set of nodes (i.e. paper, author, venue, and affiliation) and edges (e.g. author-of, published-in, and affiliated-with) enriched as follows:

- We added attributes to graph nodes and edges. For example, we enriched nodes typed as 'author' with attributes such as type, number of publications/citations, G-index [20], and

[2]The Resource Description Framework (RDF) represents a special kind of the considered information-networks (attributed graphs), i.e., in the RDF data model, graph edges can not be described by attributes.

[1]DBLP Bibliography (http://dblp.uni-trier.de/db/) downloaded in November 2011.

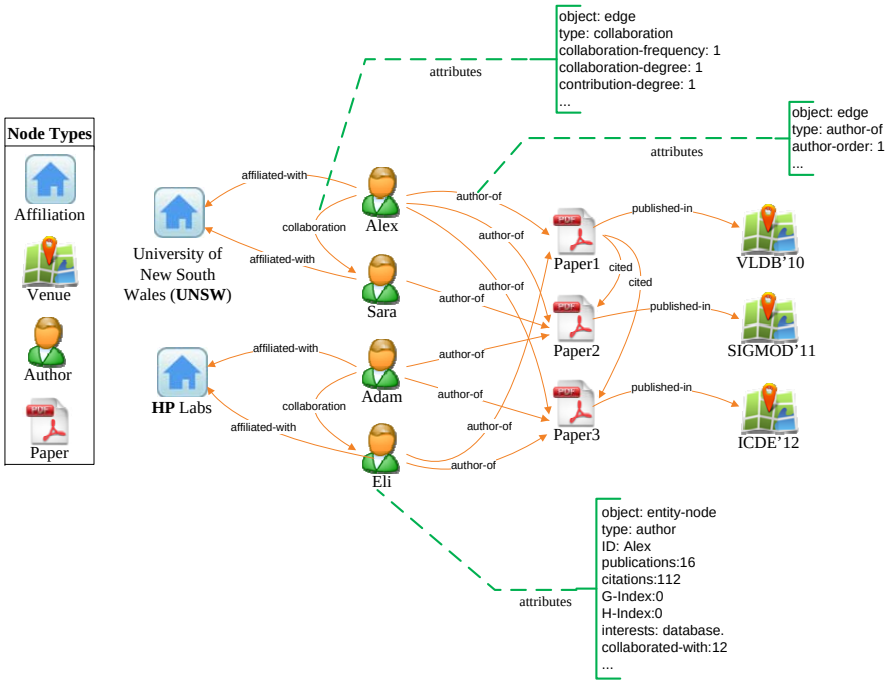


Figure 3.1: Motivating Scenario.

H-index [30]. As another example, we enriched edges typed as 'author-of' with attributes such as order of author in the paper and temporal attributes.

- We constructed edges among nodes of the graph. For example, we add an edge typed as 'collaboration' between two authors, where collaboration can be defined as a directed link between every two authors having at least one co-authored paper. The collaboration link attributes show the properties of the pairwise relation between the two authors such as: (a) *frequency of collaboration*: showing how often they have collaborated in a given period of time, e.g., in a year; (b) *degree of collaboration*: as a pairwise metric showing how the authors have collaborated in the time for example the number of the papers they have co-authored divide by all the papers every one has; (c) *mutual impact*: depicting how they have had impact on each other's publications; and *degree of contribution*: depicting what portion of community contributions i.e. papers are generated by this author.

Figure 3.1 illustrates a simplified DBLP graph. This dataset contains over 1,670,000 nodes (i.e. authors, papers, venues, and affiliations) and over 2,810,000 edges (i.e., author-of, published-in, affiliated-with, cited, and published-in). We use this dataset in the paper to demonstrate how various users can use the GOLAP framework, proposed in this paper, for supporting decision making based on multidimensional analysis of graphs. For example, we will show how an analyst can: (i) partition objects and their relationships into disjoint subsets, i.e. folder/path nodes; (ii) identify dimensions within each partition; and (iii) identify measures, with different grains (e.g. number of papers accepted in all venues or a specific conference) and types (e.g. additive and non/semi-additive measures which can be added across any/non/some dimensions).

4 Preliminaries

4.1 Data Model

We consider extending decision support on multidimensional networks by introducing a model, i.e. GOLAP, for network exploration and summarization. In [10], we proposed a graph data model to represent information networks. We reuse this data model to represent online analytical processing on attributed graphs. In particular, GOLAP data model supports: (i) uniform representation of nodes and edges; (ii) structured and unstructured entities; (iii) folder and path nodes, which can represent a network snapshot, i.e. a subgraph, from multiple perspectives and granularities. We represent entities and relationships as a directed attributed graph $G = (V, E)$ where V is a set of nodes representing entities and folder/path nodes, and E is a set of directed edges, relationships, among nodes.

Entities. An entity is a data object that exists separately and has a unique identity. Entities could be structured or unstructured. Structured entities are instances of entity types. An entity type consists of a set of attributes. Unstructured entities, are also described by a set of attributes but may not conform to an entity type. This entity model offers flexibility when types are unknown and take advantage of structure when types are known. For the sake of simplicity, we assume all unstructured entities are instances of a generic type called *ITEM*. ITEM is similar to *generic table* in [40]. Entities could be simple or composite (larger entity that can be inferred from the nodes, e.g., folder/path nodes).

A folder node contains a set of entities that are related to each other. In other words, the set of entities in a folder node is the result of a given query that require grouping graph entities in a certain way. A folder node creates a higher level node that other queries could be executed on top of it. Folders can be nested, i.e., a folder can be a member of another folder node, to allow creating and querying folders with relationships at higher levels of abstraction. A folder may have a set of attributes that describes it.

We define a path node for each query that results in a set of paths, where a *path* is a transitive relationship between two entities showing the sequence of edges from the start entity to the end. This relationship can be codified using regular expressions [10] in which alphabets are the nodes and edges from the graph. We use existing reachability approaches to verify whether an entity is reachable from another entity in the graph. Some reachability approaches (e.g. all-pairs shortest path [8]) report all possible paths between two entities. We define a path node as a triple of (V_{start}, V_{end}, RE) in which V_{start} is the start node, V_{end} is the end node and RE is the regular expression.

Folder/Path nodes can be considered as a network snapshot used to support multidimensional analysis of graphs from multiple perspectives and granularities. Folder/Path nodes may conform to an entity type and can be described by a set of attributes. The set of (related) entities in a folder/path node is the result of a given query that requires grouping graph entities based on set of dimensions coming from the attributes of: (i) graph entities; or (ii) network structures, i.e., patterns among graph entities.

Relationships. A relationship is a directed link between a pair of graph entities, which is associated with a predicate defined on the attributes of nodes that characterizes the relationship. A relationship may conform to a type and can be described by a set of attributes. A relationship can be *explicit*, such as *published-in* in 'paper *published-in* venue' in a bibliographical network. Also a relationship can be *implicit*, such as a relationship between an entity and a larger (composite) entity, e.g. folder/path nodes, that can be inferred from the nodes.

4.2 Data Elements

Cubes (Q). A cube enables effective analysis of the graph data from different perspectives and with multiple granularities. We reuse and extend the definition for graph-cube proposed in [58]. In particular, given a multidimensional network N, the graph cube is obtained by restructuring N in all possible aggregations of set of node/edge attributes A, where for each aggregation A' of A, the measure is an aggregate network G' w.r.t. A'. In order to consider both multidimensional attributes and network structures (i.e. patterns among entities) into one integrated framework for network aggregation, we define possible aggregations upon multidimensional networks using partitions. In particular, $Q = \{q_1, q_2, ..., q_n\}$ is a set of n cubes, where each q_i is a cube, a placeholder for set of partitions, and can be modeled using folder nodes. A partition can be considered as the unit of granularity, supports multi-dimensional and multi-level views over graphs, and allows evaluation of (OLAP) operations providing a natural parallelization of execution. Three types of partitions are recognized: CC/PC/Path-Partitions.

Example 1. [Correlation Condition (CC)] A *correlation condition* (CC) can be used to group related entities in a graph based on set of dimensions coming from the attributes of graph entities. In our previous work [38], we introduced the notion of a correlation condition ψ as a binary predicate defined on the attributes of data objects that allows to identify whether two or more entities (in a given graph) are potentially related. Correlation conditions can be used to partition graphs (i.e. referred in this paper as CC-Partitions) based on set of dimensions coming from the attributes of node entities. Folder nodes can be used to represent CC-Partitions. For example, Adam, an OLAP analyst, is interested in partitioning the DBLP graph into a set of related node entities having the same type. The correlation condition $\psi(node_x, node_y) : node_x.type = node_y.type$ can be defined over the attribute *type* of two node entities $node_x$ and $node_y$ in the DBLP graph. This predicate is true when $node_x$ and $node_y$ have the same type value and false otherwise. Related node entities will be stored in folders, where each folder can conform to an entity type and described by a set of attributes. Figure 4.1-A represents the result of this example.

Example 2. [Path Condition (PC)] A *path condition* (PC) can be used to group related entities in a graph based on set of dimensions coming from the attributes of network structures, patterns, among graph entities. A path condition ϕ is a binary predicate defined on the attributes of a path (see Section 4.1) that allows to identify whether two or more entities (in a given graph) are potentially related through that path. The (transitive) relationship, in a path, can be codified

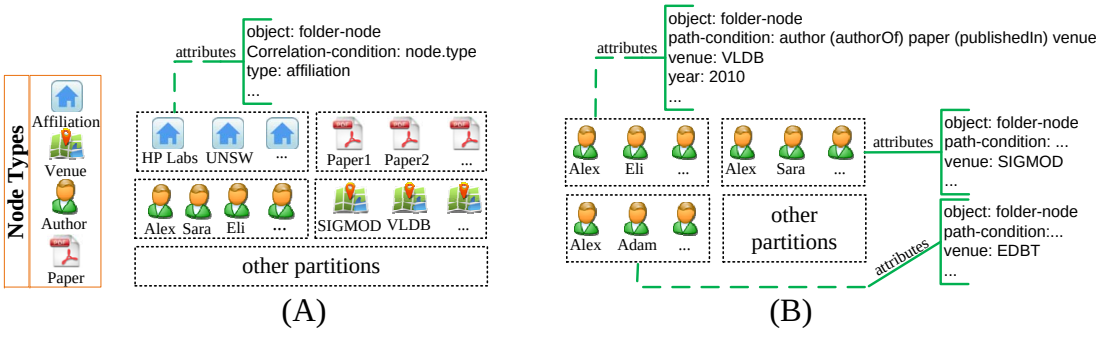


Figure 4.1: Examples of folder partitions: (A) result of Example 1; and (B) result of Example 2.

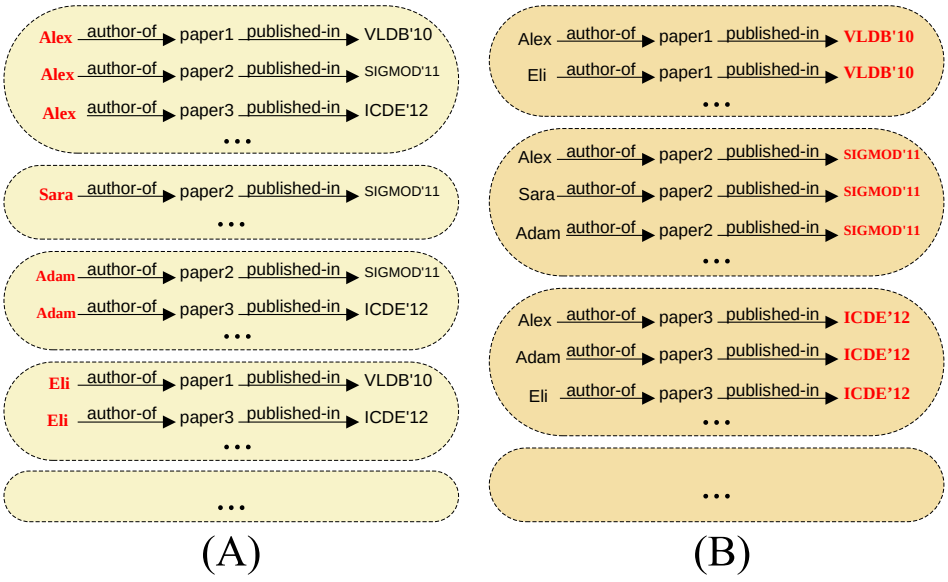


Figure 4.2: Result of Example 3 grouped by: (A) authors; and (B) venues.

using regular expressions[1] in which alphabets are the nodes and edges from the graph (details can be found in our previous work [10]). Path conditions can be used to partition graphs (i.e. referred in this paper as PC-Partitions) based on set of dimensions coming from the attributes of node and edge entities. For example, Adam is interested in partitioning the DBLP graph into a set of related authors having at least one paper published in a specific venue. In other words, Adam is interested in: (i) creating partitions for a set of related authors; and (ii) adding authors to related partitions if: there exists the path '$author \xrightarrow{(authorOf)} paper \xrightarrow{(publishedIn)} venue$' between the author and the venue. In this case, correlation conditions cannot be used to partition the graph as we need to apply conditions not only on graph entities but also on the relationships among them. The path condition $\phi(node_{start}, node_{end}, RE)$ can be defined on the existence of the path codified by the regular expression RE:$[author(authorOf)paper(publishedIn)venue]$ between starting node, $node_{start}$, and ending node, $node_{end}$. This predicate is *true* if the path exists, and *false* otherwise. Related authors satisfying the path condition, will be stored in folders. Figure 4.1-B represents the result of this example. Notice that PC-Partitions will contain set of nodes, will be modeled using folders, and further operations will apply on node members.

Example 3. [Path-Partitions] There are cases where OLAP analysts may be interested in partitioning the graph into set of related paths. Path conditions can be used to discover set of paths and store them in a path node. Further operations can be applied to constructed Path-Partitions. For example, Adam is interested in partitioning the DBLP graph into a set of related paths having the pattern 'RE: $author(authorOf)paper(publishedIn)venue$'. Adam can create partitions by grouping set of related paths , e.g. group by authors (Figure 4.2-A) or venues (Figure 4.2-B). These partitions can be modeled using path nodes. Adam may apply further operations to constructed partitions. For example, considering Figure 4.2-A, he can apply op-

[1] A regular expression supports optional elements (?), loops (+,*), alternation (|), and grouping ((...)). Also, the symbols | and / are interpreted as logical OR and composition respectively (details can be found in our previous work [10]).

erations to calculate G-index, H-index, or number of publications/citations for all (or specific) authors. As another example, considering Figure 4.2-B, it is possible to calculate quality metrics for venues by analyzing related papers and/or authors quality metrics.

Dimensions (D). Dimensions can be defined as perspectives used for looking at the data. In particular, $D = \{d_1, d_2, ..., d_n\}$ is a set of n dimensions, where each d_i is a dimension name. Each dimension d_i is represented by a set of elements (E) where elements are the nodes and edges of the graph. In particular, $E = \{e_1, e_2, ..., e_m\}$ is a set of m elements, where each e_i is an element name. Each element e_i is represented by a set of attributes (A), where $A = \{a_1, a_2, ..., a_p\}$ is a set of p attributes for element e_i, and each a_i is an attribute name. A dimension d_i can be considered as a given query that require grouping graph entities in a certain way. Correlation conditions and path conditions can be used to define such queries. The result of this query can be stored in folder/path nodes, see Examples 2 and 3.

Cells (C). A dimension uniquely identify a subgraph within each partition, which we call a *cell*. In particular, $C = \{c_1, c_2, ..., c_n\}$ is a set of n cells, where each c_i is a cell name. Each cell can be constructed using GOLAP operations (see *operations* definition). For example, considering the motivating scenario, we may be interested in a set of dimensions coming from: (i) the attributes of graph nodes, e.g. *authors* with specific g-index or h-index; or (ii) the attributes of graph nodes and edges, e.g. *authors* who published in specific venues or *authors* affiliated with universities in Australia during a specific time period. In order to identify cells, dimensions may have *levels* used for drilling down/up, where levels enable visiting the general/detailed view of dimensions. For example, it is important to see if the number of publications for a specific author (or group of related authors) are higher in a particular year, or drill down to see if they were higher in a particular part of the year (e.g. a specific month).

Measures (M). Dimensions can be used as an index in order to analyze measures. A measure can be considered as numerical and computational attributes of dimensions' elements, i.e., nodes and edges of the graph. In particular, $M = \{m_1, m_2, ..., m_n\}$ is a set of n measures, where each m_i is a measure name. For example, in the motivating scenario, number-of-publications can be considered as a measure for the element 'author' (which is a node entity) in a specific dimension. As another example, collaboration-frequency can be considered as a measure for the element 'collaboration' (which is an edge entity) in a specific dimension. Measures can be calculated by applying operations to multidimensional graph data, where operations can support UPSERT and UPDATE semantics (see *operations* definition). Three types of measures can be recognized:

- *entity attributes*: (i) nodes: The attribute for existing graph nodes, i.e. entities/folders/paths, can be considered as measure m_i. The value for existing attributes can be updated or new attributes can be added to nodes as the result of a GOLAP operation. For example, the value for the attribute 'citations' of an author can be updated/inserted during a GOLAP operation; and (ii) edges: The attribute value for existing graph edges can be considered as measure m_i. The value for existing attributes can be updated or new attributes can be added to edges as the result of a GOLAP operation. For example, the value for the attribute 'collaboration-frequency' of a collaboration edge between two authors can be updated/inserted during a GOLAP operation.

- *aggregated nodes*, are subgraphs including set of related nodes and relationships among them which can be considered as measure m_i. For example, considering Figure 3.1, OLAP analysts may be interested in the collaborative relationship between researchers affiliated

with HP Labs and the University of New South Wales (UNSW), e.g. see Example 7. Folder and path nodes are examples of aggregated nodes, can be added to graphs during a GOLAP operation.

- *inferred edges*, are new edges which can be added between two nodes in the graph as a result of a GOLAP operation. For example, collaboration between two authors can be calculated and can be added as weighted edge between two authors in the graph.

Operations (O). Typical operations on data cubes are: (i) roll-up: to aggregate data by moving up along one or more dimensions; (ii) drill-down: to disaggregate data by moving down dimensions; and (iii) slice-and-dice: to perform selection and projection on snapshots. Such operations are supported to explore different multidimensional views and allow interactive querying and analysis of the underlying data. Consequently, operations can be used for describing a computation on cells and can be ordered based on the dependencies between cells. In particular, $O = \{o_1, o_2, ..., o_n\}$ is a set of n operations, where each o_i is a operation name. Operations support UPSERT and UPDATE semantics. In order to provide a natural parallelization of execution, each operation should be evaluated independently for each partition. In GOLAP, operations can be divided into two categories:

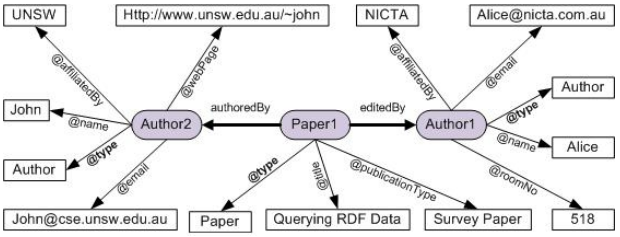
- *assignments*: are defined to apply operations on entity attributes. Assignments support correlation between the left side (which designates target cells) and right side (which contains expressions involving cells or ranges of cells within the partition), i.e., to simulate the effect of multiple joins and UNIONs using a single access structure. For example, it is possible to apply operations on number of publications (for different group) of authors in order to rank each author (see Example 4).

- *functions*: are defined to apply set of related operations on network structures among entities. A function can be considered as a portion of SPARQL patterns (see Section 5) used to apply operations on the constructed partitions. For example, functions can be used to aggregate/disaggregate authors, to calculate measures such as G-index, H-index, and number-of-publication of authors, and to calculate collaboration-frequency between two authors (see Examples 5, 6, 7, and 8).

5 A Query Language for OLAP on Graphs

Querying OLAP graphs needs a graph query language that not only supports primitive graph queries but also is capable of: (i) partitioning graphs in order to provide a natural parallelization of execution. For example, a query can be used to partition the DBLP dataset into a set of related nodes (Example 1 and Example 2), or set of related paths (Example 3); (ii) identifying cells by applying further queries to constructed partitions, e.g. using a dimension to (uniquely) identify a subgraph within each partition. For example, consider that we have partitions for authors according to their affiliation, and in each partition (e.g. authors of the University of New South Wales) we are looking for group of authors having at least one paper published in a database venue; (iii) applying operations on partitions and cells; and (iv) applying existing graph mining algorithms to OLAP graphs for further analysis. For example, we may need to apply existing reachability algorithms or frequent pattern discovery algorithm to discover patterns.

We proposed FPSPARQL [10], a graph query processing engine, which is a Folder-Path enabled extension of SPARQL [41]. FPSPARQL supports primitive graph queries, constructing folder/path nodes, applying further queries to constructed folder/path nodes, and applying external tools and algorithms to graph. In particular, FPSPARQL supports two levels of queries:



(a) Sample RDF Graph.

```
SELECT  ?Z
WHERE { ?X  @hasTitle "Querying RDF Data".
        ?X  @publicationType "Survey Paper".
        ?X  authoredBy ?Y.
        ?Y  @webPage ?Z. }
```

(b) Sample SPARQL query.

Figure 5.1: Representation of a sample RDF graph and a SPARQL query.

(i) Graph-level Queries: at this level we use SPARQL to query graphs; and (ii) Node-level Queries: at this level we propose an extension of SPARQL to construct and query folder nodes and path nodes.

Graph-level Queries. SPARQL is an RDF query language, standardized by the World Wide Web Consortium, for semantic web. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries. The results of SPARQL queries can be results sets or RDF graphs. A basic SPARQL query has the form:

```
select ?variable1 ?variable2 ...
where {
  pattern1.
  pattern2.
  ...
}
```

Each pattern consists of *subject*, *predicate* and *object*, and each of these can be either a variable or a literal. The query specifies the known literals and leaves the unknowns as variables. To answer a query we need to find all possible variable bindings that satisfy the given patterns. We use the '@' symbol for representing attribute edges and distinguishing them from the relationship edges between graph nodes. Figure 5.1(b) depicts a sample SPARQL query over the sample RDF graph of Figure 5.1(a) to retrieve the web page information of the author of a book chapter with the title "Querying RDF Data".

Node-level Queries. Standard SPARQL querying mechanisms is not enough to support querying needs of FPSPARQL and its data model. In particular, SPARQL does not support folder nodes and querying them natively and such queries needs to be applied to the whole graph. In addition, querying the result of a previous query becomes complex and cumbersome, at best. Also path nodes are not first class objects in SPARQL [8, 31]. In [10], we extended SPARQL to support node-level queries to satisfy specific querying needs of our data model. Node-level queries in FPSPARQL include two special constructs: (a) CONSTRUCT queries: used for constructing folder and path nodes, and (b) APPLY queries: used to ease the requirement of applying queries on folder and path nodes.

Following we extend FPSPARQL for querying and analyzing GOLAP networks.

5.1 FPSPARQL Extensions for OLAP on Graphs

Online analytical processing on graphs requires dividing objects and relationship among them into partitions, dimensions, and measures. To model that, we extend FPSPARQL by introducing the 'GOLAP' clause. This command is used to identify partitions, dimensions, and measures on graphs. A basic GOLAP query looks like this:

```
Select <measures>
Where{
 # <existing parts of a query block>
 GOLAP{
   ?analytic [@CC|@PC|@Path-Partition] <identify partitions>.
   ?analytic @dimension <identify dimensions>.
   ?analytic @measure <identify measures>.

   <operation>; <operation>; ... <operation>;
 }
}
```

The GOLAP command is evaluated after aggregations but before ORDER BY clause. The variable ?ANALYTIC is defined to specify partitions, dimensions, and measures, where:

- *Partitions*: can be considered as: (i) CC-Partition: folder nodes which are constructed according to a correlation condition (CC), e.g., see Example 4; (ii) PC-Partition: folder nodes which are constructed according to a path condition (PC), e.g., see Example 5; and (iii) PATH-Partition: path nodes which are constructed according to a regular expression, e.g., see Example 8.
- *Dimensions*: can be defined on the attributes of set of: (i) node elements, and can be used in assignments, e.g. see Example 4; or (ii) node and edge elements, and can be defined and used in functions, e.g. see Examples 5, 6, 7, and 8.
- *Measures*: are defined to identify expressions computed by GOLAP clause, e.g., Examples 4 to 8.

GOLAP operations can be defined in two ways:

1. *Assignments*: are defined to apply operations on the attributes of node elements. Assignment statement will update or upsert the value of a cell. Operations support the correlation between the left side (which designates target cells) and right side (which contains expressions involving cells or ranges of cells within the partition) of assignment (see Example 4).
2. *Functions*: are defined to apply operations on edge attributes, aggregated nodes, and inferred edges measures. A function is a portion of SPARQL patterns used to apply operations on the constructed partitions (i.e. partition members and relationships among them). For example, using functions, it will be possible to establish new (attributed) edges between pair of entities as a result of an operation (see Examples 5, 6, 7, and 8).

By default, the evaluation of operations occurs in the order of their dependencies. However, there are scenarios in which sequential ordering of evaluation is desired. We provide an explicit processing option, i.e. SEQUENTIAL, for that as in: $SEQUENTIAL(... < operation > ...)$. Following we explain the extension for online analytical processing on graphs through examples.

Example 4. [**nodes attributes**] Adam is interested in partitioning the DBLP graph into a set of authors having same interests. We consider 'interest' as an enumerated type consisting of a set of named values (i.e. elements) such as database, business process, and cloud. Adam is interested in applying following operations on constructed partitions:

1. Update the value for the 'rank' attribute (i.e. the value for this attribute is a real number in a fixed range [0,M] where M means the highest rank) of the authors who has:
   - more than 200 publications to '6';
   - more than 200 publications and more than 1000 citations to 50% higher than the authors having same number of publications but less than 1000 citations;
   - between 100 and 200 publications to '4';
   - between 50 and 100 to '2';
   - less than 50 to '1';
2. Insert a new attribute (i.e. 'contribution-degree') for authors within each partition. For each partition, this attribute will be calculated from division of number of publications for an authors into the sum of publications of all authors in the partition (see Section 3).

In this example, all the measures come from the attributes of nodes in the graph. Following is the FPSPARQL query for this example.

```
1  Select ?ar, ?cd
2  Where
3   {
4    GOLAP{
5     ?analytic  @CC-Partition 'node[type="author"].interest'.
6     ?analytic  @dimension 'publications as ?ap, citations as ?ac'.
7     ?analytic  @measure 'rank as ?ar, contribution-degree as ?cd'.
8     #1
9     update ?ar[?ap >= 200]= 6;
10    update ?ar[?ap >= 200 AND ?ac >= 1000] =
11                          ?ar[?ap > 200 AND ?ac < 1000]*1.5;
12    update ?ar[200 > ?ap >= 100]= 4;
13    update ?ar[100 > ?ap >= 50]= 2;
14    update ?ar[?ap < 50]= 1;
15    #2
16    upsert ?cd[*] = ?ap / ?sum_publications;
17   }
18  AGGREGATE { (?sum_publications, SUM, {?ap} ) }
19 }
```

In this example, the variable ?ANALYTIC (lines 5 to 7) is used to define partitions, dimensions, and measures. The predicate @CC-Partition (line 5) partitions the DBLP graph into a set of related nodes, i.e., set of authors having same interests where constructed partitions will be stored in a set of folder nodes. The NODE keyword (line 5) helps in partitioning the graph by filtering graph nodes through set of nodes attributes and their values in the bracket, e.g. in 'node[type="author"]', and selecting an attribute for partitioning the graph, e.g. in

'node[type="author"].interest'. In particular, line 5, will partition the graph into set of authors having same interest. The predicate @DIMENSION (line 6) used to define dimensions to apply further operations on constructed partitions. The predicate @MEASURE (line 7) used to define measures to be updated/upserted. In order to write operations in a simple way we support aliases (using AS statement) for dimensions and measures, e.g., ?*ap* in 'publications as ?ap'.

The Keyword UPDATE/UPSERT used to update/upsert a measure in an operation. Evaluation of operations will apply independently for each partition providing a natural parallelization of execution. An assignment can designate a single object reference (e.g. '[author-id = 2]' which designates an author whose ID is 2) or set of objects (e.g. '[publications >= 200]' which designates set of authors who has more than 200 publications). Assignments support the correlation between the left side and right side of assignment, e.g., in line 10 and 11. Notice that to remove unnecessary computations, assignments whose results are not referenced in outer blocks will be removed automatically.

To calculate contribution-degree for each author we use aggregate functions. In order to support aggregation functions in FPSPARQL, we implement the extension proposed in C-SPARQL [7]. Aggregation clause will be added at the end of the query, and have the following syntax:

```
AggregateClause -->
   ( 'AGGREGATE { (' var ',' Function ',' Group ')' [Filter] '}' )*
Function -->
   'COUNT' | 'SUM' | 'AVG' | 'MIN' | 'MAX'
Group    -->
   var | '{' var ( ',' var )* '}'
```

Using AGGREGATE statement we calculate sum of publications of all authors in the partition and assign this value to the variable '?sum_publications' (line 18). Notice that all aggregates are computed before evaluation of operations so they are available for the operations. Then we calculate contribution-degree, '?cd', and add it as a new attribute for all author, '?cd[*]' (line 16).

Example 5. [inferred edges] Adam is interested in partitioning the DBLP graph into a set of related authors collaborating on (specific) papers. To achieve this, set of dimensions coming from the attributes of authors, papers and the relationship among them should be analyzed. Similar to Example 2, a path-condition can be used to partition authors. After partitioning, Adam is interested to establish a pairwise 'collaboration' edge between authors in each partition. This edge may have some attributes, where attributes can be calculated in an operation. For example, considering Figure 3.1, authors (e.g. Alex and Sara) collaborating on a paper will be correlated with a *collaboration* edge having attributes such as *collaboration-frequency*, *collaboration-degree*, and *contribution-degree*. These attributes are defined in the motivating scenario.

This example represents, how the query language is able to establish an edge between two members of a partition as a result of a GOLAP operation. As future work, we will show that how this feature can be used for enriching crowd computing graphs ,e.g., establishing weighted edges between requesters and workers in a crowdsourcing[1] graph. Following is the FPSPARQL query for this example.

[1]Crowdsourcing is the act of sourcing tasks traditionally performed by specific individuals to a group of people or community (crowd) through an open call.

```
1  Select ?m, ?edge, ?n
2  Where{
3   # defining path condition
4   ?path-condition @regular-expression '?author (?authorOf) ?paper'.
5   ?path-condition @groupBy ?paper.
6   ?path-condition @partition-item 'distinct ?author'.
7
8   # defining @regular-expression variables
9   ?author @isA entityNode. ?author @type author.
10  ?authorOf @isA edge. ?authorOf @type author-of.
11  ?paper @isA entityNode. ?paper @type paper.
12
13   GOLAP{
14    ?analytic  @PC-Partition ?path-condition.
15    ?analytic  @dimension '?m, ?n'.
16     #dimensions(s) are defined in the function!
17    ?analytic  @measure '?edge'.
18     #measure(s) are defined in the function!
19
20    function.F1;
21   }
22 }
23
24 functions{
25  F1{
26   ?m @isA entityNode. ?m @type author. ?m @name ?m_name.
27   ?n @isA entityNode. ?n @type author. ?n @name ?n_name.
28   correlate{
29    (?m,?n,?edge,FILTER(?m_name <> ?n_name))
30    ?edge @isA edge. ?edge @type 'collaboration'.
31    ?edge @collaboration-frequency '0'.
32    ?edge @collaboration-degree '0'.
33    ?edge @contribution-degree '0'.
34   }
35  }
36 }
```

In this example, the predicate @PC-Partition (line 14) is used to partitions the DBLP graph into a set of related authors, where each partition contains authors of a specific paper. This condition is generated through the regular expression '$author(authorOf)paper$' (line 4), where variable ?PATH-CONDITION is used to define the path-condition attributes. Also set of predicates used to define the regular expression (@REGULAR-EXPRESSION in line 4), grouping selected paths (@GROUPBY in line 5), and defining the items to be added to constructed partitions (@PARTITION-ITEM in line 6). Moreover, The DISTINCT keyword can be used to add distinct (different) values to the partition. The second block of codes in this example (lines 9 to 11), defines the elements of regular expression such as ?AUTHOR, ?AUTHOROF, and ?PAPER.

To construct an attributed edge between two nodes in partitions we use functions. The FUNCTIONS keyword (line 24) is used to define a block of functions. Each function defined by

a name (e.g. function F1) and a block of SPARQL patterns. Defined functions can be called using FUNCTION keyword (line 20) following by a full stop and function name (e.g. 'function.F1'). In this example, function F1 defined to establish a collaboration edge among authors. In our previous work [10], we introduced CORRELATE statement to establish a directed edge between two nodes in a graph. A basic correlation condition query looks like this:

```
correlate {
   (entity1, entity2, edge1, condition)
   pattern1.
   pattern2.
   ...
}
```

As a result, $entity_1$ will be correlated to $entity_2$ through a directed edge $edge_1$ if the condition evaluates to true. Patterns can be used for specifying the edge attributes. In function F1 (lines 25 to 35), variables $?m$ and $?n$ represent authors. The condition in *correlate* statement (i.e., $?m_{name} <> ?n_{name}$) makes sure that only two different authors will be connected. For simplicity reason, in this example we assign a constant value for collaboration edge attributes. Next example shows how attributes of the collaboration edge can be calculated dynamically.

Example 6. **[edges attributes]** Adam is interested in calculating the degree of collaboration in Example 5. In section 3, degree of collaboration defined as a pairwise metric showing how the authors has collaborated in time, e.g., the number of papers they have coauthored divided by all the papers every one has. Notice that this attribute may have different values over time. Following, we revise function F1 in Example 5, in order to calculate the collaboration degree between authors.

```
1  F1{
2   ?m @isA entityNode. ?m @type author. ?m @name ?m_name.
3   ?n @isA entityNode. ?n @type author. ?n @name ?n_name.
4   ?m @publications ?pubs.
5
6   ?edge @isA edge. ?edge @type 'collaboration'.
7   ?edge @numOfCoauthoredPapers ?numCoPapers.
8   ?edge @collaboration-degree ?clbDegree.
9
10  optional{
11   correlate{
12    (?m,?n,?edge,FILTER(?m_name <> ?n_name))
13    BIND (1 AS ?numCoPapers). # in SPARQL, BIND assign a value to a variable
14    BIND (?numCoPapers/?pubs AS ?clbDegree)
15   }
16  } FILTER NOT EXISTS { ?m ?edge ?n }
17
18  optional{
19   BIND (?numCoPapers+1 AS ?numCoPapers).
20   BIND (?numCoPapers/?pubs AS ?clbDegree)
21  } FILTER EXISTS { ?m ?edge ?n }
22 }
```

In this example, partitions will be constructed similar to Example 5, where each partition contains authors of a specific paper. Considering that the operations are evaluated for one partition at a time, to calculate the number of coauthored papers between two authors it will be enough to follow two steps: (a) if the collaboration edge does not exist between authors, construct the edge and set the coauthored-papers attribute to 1; and (b) if the collaboration edge exists, increase the value for coauthored-papers attribute by 1. Variables ?m and ?n represent authors in the partitions and defined in lines 2 to 4, and variable ?*pubs* represents the number of publications for author ?m. The collaboration edge, i.e., ?*edge*, is defined in lines 6 to 8, where variable ?*numCoPapers* represents number of coauthored papers between authors ?m and ?n. The variable ?*clbDegree* (line 8) represents the collaboration degree between authors ?m and ?n and will be calculated by '?numCoPapers/?pubs', i.e., number of papers they have coauthored divided by all the papers every one has.

There are two optional statements used in this function. The first one (lines 10 to 16) used to construct the collaboration edge ?*edge* between authors ?m and ?n if the edge does not exists between them. In this case, the edge will be constructed (line 13), the number of coauthored papers will be set to 1 (line 13), and the collaboration degree will be calculated (line 14). The second optional statement (lines 18 to 21) used to updated the number of coauthored papers and recalculating the collaboration degree if the edge exists between authors ?m and ?n. Notice that in SPARQL, the 'EXISTS' and 'NOT EXISTS' statements are used based on testing whether a pattern exists/not-exists in the graph. Moreover, in SPARQL, the BIND statement allows a value to be assigned to a variable in a group graph pattern.

Example 7. [aggregated nodes] Adam is interested in the collaborative relationship between researchers affiliated with affiliations, e.g., HP Labs and UNSW. To achieve this, he partitioned the graph into a set of related authors affiliated with specific affiliations by setting the regular expression to 'author (affiliated-with) affiliation'. Then he set the group-by attribute to 'affiliation' and put related authors in partitions. This example shows how partitions can be stored (e.g. as a folder node in this example) and added to the graph as a result of a GOLAP query. Notice that such aggregated nodes can have set of descriptive attributes. This way Adam will be enable to construct relationships (e.g. 'collaboration' edge) among aggregated nodes in the graph. Following is the FPSPARQL query for this example.

```
1  Select ?aff_title
2  Where{
3   # defining path condition
4   ?path-condition @regular-expression '?author (?affiliated-with) ?affiliation'.
5   ?path-condition @groupBy ?affiliation.
6   ?path-condition @partition-item 'distinct ?author'.
7
8   # defining @regular-expression variables
9   ?author @isA entityNode. ?author @type author.
10  ?affiliated-with @isA edge. ?affiliated-with @type affiliated-with.
11  ?affiliation @isA entityNode. ?affiliation @type affiliation.
12  ?affiliation @title ?aff_title. #e.g., UNSW or HP Labs.
13
14   GOLAP{
15    ?analytic  @PC-Partition ?path-condition.
16    ?analytic  @dimension '?author, ?aff_title'.
17     #dimensions(s) are defined in the function!
```

```
18    ?analytic  @measure '?folderNode'.
19     #measure(s) are defined in the function!
20
21    function.F1;
22   }
23 }
24
25 functions{
26  F1{
27   BIND (?folderNode as ?aff_title)  #?aff_title is defined in line 12
28   fconstruct ?folderNode
29    select * where {
30    ?folderNode @description 'set of ...'.
31    ?folderNode @numberOfAuthors ?authorsCount.
32    # other patterns.
33   }
34   AGGREGATE { (?authorsCount, COUNT, {?author} ) }
35  }
36 }
```

In this example, the predicate @PC-Partition (line 15) is used to partitions the DBLP graph into a set of related authors affiliated with specific affiliations. This condition is generated through the regular expression '$author(affiliated - with)affiliation$' (line 4). The variable ?PATH-CONDITION is used to define the path-condition attributes, i.e., @REGULAR-EXPRESSION, @GROUPBY, and @PARTITION-ITEM. The second block of codes (lines 9 to 11), defines the elements of regular expression such as ?AUTHOR, ?AFFILIATED-WITH, and ?AFFILIATION. Then the GOLAP statement will partition the graph. Finally the function F1 will apply on all partition. As a result, each partition will be stored in a folder node and will be added to the graph as an aggregated node.

In [10] we introduced the FCONSTRUCT command which is used to group a set of related entities or folders. A basic folder node construction query looks like this:

```
fconstruct <Folder_Node Name>
[ select ?var1 ?var2 ... | (Folder_Node1 Name, Folder_Node2 Name, ...) ]
where {
   pattern1.
   pattern2.
   ...
}
```

A query can be used to define a new folder node by listing folder node name and entity definitions in the *fconstruct* and *select* statements, respectively. Also a folder node can be defined to group a set of folder nodes. A set of user defined attributes for this folder can be defined in the *where* statement. In this example we use FCONSTRUCT command in function F1 in order to store each partition as a folder node and add them to the graph. We set the name of the folder node ($?folderNode$) as the affiliation title ($?aff\_title$) by binding its value to the folder to be constructed. For example, if function F1 apply on the partition which contains set of authors affiliated with UNSW, then the variable $?aff\_title$ will contain 'UNSW' (see line 12 and 27), and consequently the name of the folder to be constructed will be 'UNSW'. Notice that

the folder node can have set of descriptive attributes, e.g., description and number of authors (line 30 and 31). The value for these attributes can be calculated dynamically, e.g., number of authors.

Next, Adam can use a function, similar to example 5, to construct relationships (e.g. 'collaboration' edge with attributes such as *collaboration-frequency*, *collaboration-degree*, and *contribution-degree*) between aggregated nodes (e.g. folder nodes in this example). Recall from [10] that we introduced the PCONSTRUCT command to construct path nodes. In particular, in scenarios similar to Example 7, FCONSTRUCT can be used to store CC-Partitions and PC-Partitions in folder nodes and PCONSTRUCT can be used to store Path-Partitions in path nodes. The syntax for using PCONSTRUCT is similar to the FCONSTRUCT.

Example 8. [path partitions] Adam is interested in partitioning the DBLP graph into a set of related paths having the pattern 'RE: $author(authorOf)paper(publishedIn)venue$', and group by authors. Figure 4.2-A illustrates such partitions. As next step, Adam is interested in:

- Update the number of publications for each author. This query can be done by counting papers in each partition.
- Update the number of citations for each author. This can be done by calculating the summation of all papers' citations.
- calculate ERA (Excellence in Research for Australia) rank for each author. For example, consider that in ERA ranking, papers published in venues ranked: (i) 'A*' have 4 points; (ii) 'A' have 1 point; (iii) 'B' have 0 point; and (iv) 'C' have -1 point.

Following is the FPSPARQL query for this example.

```
1  Select ?pub, ?paperCitation, ?authorCitation, ?ERA
2  Where{
3   # defining path condition
4   ?path-node @regular-expression
5              '?author (?authorOf) ?paper (?publishedIn) ?venue'.
6   ?path-node @groupBy ?author.
7
8   # defining variables used in the regular expression for the path-node
9   ?author @isA entityNode. ?author @type author.
10  ?authorOf @isA edge. ?authorOf @type author-of.
11  ?paper @isA entityNode. ?paper @type paper.
12  ?publishedIn @isA edge. ?publishedIn @type published-in.
13  ?venue @isA entityNode. ?venue @type venue.
14  ?paper @citations ?paperCitation
15  ?author @publication ?pub.
16  ?author @citation ?authorCitation.
17  ?author @ERA_Ranking ?ERA.
18  ?venue @isA entityNode. ?venue @type venue.
19
20   GOLAP{
21    ?analytic  @Path-Partition ?path-condition.
22    ?analytic  @dimension '?author, ?paper, ?venue'.
23    #dimensions are defined in lines 9 to 18!
```

```
24    ?analytic  @measure '?pub, ?paperCitation, ?authorCitation, ?ERA'.
25    #measures are defined in lines 9 to 18!
26
27    function.F1;
28    function.ERA;
29   }
30 }
31
32 functions{
33  F1{
34   #1-update number of publications for each author.
35   update ?pub[*] = ?numberOfPapers;
36
37   #2-update number of citations for each author.
38   update ?authorCitation[*] = ?allCitations;
39
40   AGGREGATE { (?numberOfPapers, COUNT, {?paper} ) }
41   AGGREGATE { (?allCitations, SUM, {?paperCitation} ) }
42  }
43
44  ERA{
45   #4-calculating ERA ranking
46   update ?ERA[*] = (?numOfTopA * 4)+(?numOfA * 1)-(?numOfC * 1);
47
48   AGGREGATE { (?numOfTopA, COUNT, {?paper} )
49     FILTER (?paper published-in ?venue. ?venue @Rank = 'A*') }
50   AGGREGATE { (?numOfA, COUNT, {?paper} )
51     FILTER (?paper published-in ?venue. ?venue @Rank = 'A') }
52   AGGREGATE { (?numOfC, COUNT, {?paper} )
53     FILTER (?paper published-in ?venue. ?venue @Rank = 'C') }
54  }
55 }
```

In this example, parameter @PATH-Partition (line 21) is used to partition the graph into set of path nodes. The regular expression for path nodes is defined in the parameter @REGULAR-EXPRESSION (line 4 and 5). Predicate @GROUPBY is used to group discovered paths by authors (line 6). The second block of code (lines 8 to 18) defines the variables used in the regular expression. The GOLAP statement (lines 20 to 29) defines dimensions and measures to be used in functions F1 and ERA. In function F1, aggregates used to calculate the number of papers (line 40) and sum of citations (line 41) for the authors. The assignment in line 35 will update the number of publications, '?pub', for all authors, '?pub[*]'. The assignment in line 38 will update the number of citations, '?authorCitation', for all authors, '?authorCitation[*]'. In function ERA, aggregates used to calculated number of papers published in venues ranked 'A*', 'A', and 'C'. Note that, venues ranked as B will not be calculated as they have zero points. The assignment in line 46 will update the ERA ranking, '?ERA', for all authors, '?pub', for all authors, '?ERA[*]'.

5.2 Query Optimization

The partitioning of the graph provides an obvious way to parallelize OLAP operations on graph partitions and provide scalability. In proposed query language, partitioning can be done through executing an SPARQL query. The key optimizing issue here is join ordering, which we addressed in [10]. For efficient processing of partitions, we apply techniques for indexing RDF triples proposed in [39]. Moreover, we use a path-based indexing approach (i.e., gIndex [56]) for partitioning the graph based on patterns among graph entities (similar to Examples 2 and 3). We use gIndex to build path indices, in order to help processing path conditions and constructing path partitions. Moreover, to eliminate cycles, we applied the cycle elimination technique proposed in [25, 59].

FPSPARQL supports *aggregate* queries and *keyword* search queries [10]. To help users conduct partitioning on graphs and to enhance the capability of the keyword search technique on triple tables, we develop the aggregate keyword search method which finds aggregate groups of entities jointly matching a set of query keywords, i.e. both for folder and path nodes. Moreover, for efficient access to single cells we built a partition level hash access structure. In particular, data is hash partitioned on '@CC-Partition', '@PC-Partition', and '@Path-Partition' parameters. Moreover, for CC-Partitions and PC-Partitions, a hash table will be built on the dimensions. For Path-Partitions, two hash tables will be built, one for nodes in path partitions and the other for relationship edges. Consequently, we avoid spilling to disk for evaluating the operations: the partitions will be kept in memory and the operations will evaluated for one partition at a time.

From the operation dependencies point of view, we have two types of operations: assignments and functions. The order of evaluation of assignments is determined from their dependency graph, where the dependency graph is a directed graph representing dependencies of several assignments towards each other. For example, given a several assignments like "i-Index = *sum*(publications)*c-measure; c-measure = *avg*(citations)", then 'i-Index' depends on 'c-measure' which should be calculated before 'i-Index'. We derive an evaluation order, or the absence of an evaluation order, that respects the given dependencies from the constructed dependency graph. Moreover, we use the dependency graph to identify the operations that can be *pruned*, e.g., the evaluation of an assignment becomes unnecessary when the cells it updates are not used in the evaluation of other assignments. Note that, for functions, we assume that they will independently apply on constructed partitions.

6 Architecture, Implementation and Experiments

6.1 Architecture

Figure 6.1 illustrates FPSPARQL graph processing architecture which consists of following components:

1. *Graph Loader*: Input graph can be in the form of RDF, N3 (or Notation3, is a W3C standard and shorthand non-XML serialization of RDF models), or XML. We developed a workload-independent physical design by developing a *loader* algorithm. This algorithm is responsible for: (i) validating the input graph; (ii) generating the relational representation of triple store, for manipulating and querying entities, folders, and paths; and (iii) generating powerful indexing mechanisms.

2. *Data Mapping Layer*: is responsible for creating data element mappings between semantic web technology (i.e. Resource Description Framework) and relational database schema.

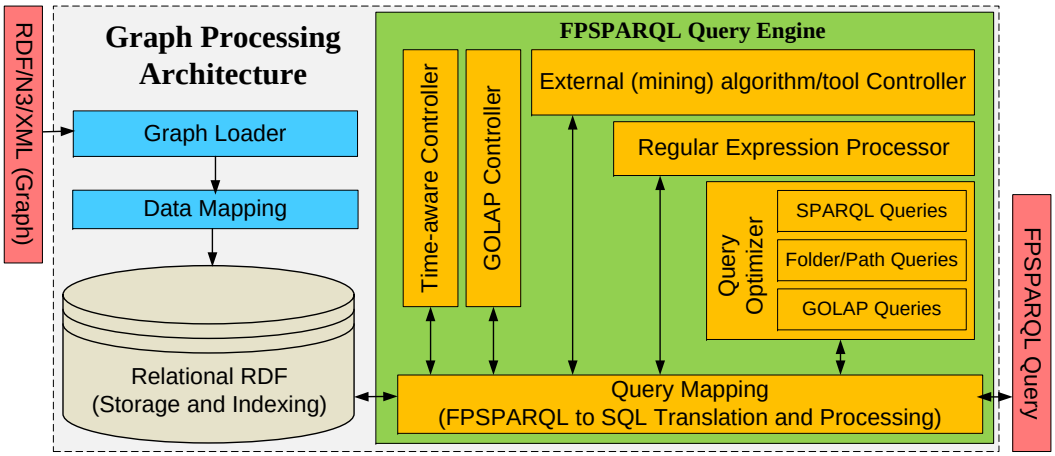


Figure 6.1: FPSPARQL graph processing architecture.

3. *Query Mapping Layer*: is consist of a FPSPARQL parser (for parsing FPSPARQL queries based upon the syntax of FPSPARQL) and a schema-independent FPSPARQL-to-SQL translation algorithm. This algorithm consists of:

   - *SPARQL-to-SQL Translation Algorithm.* We implemented a SPARQL-to-SQL translation algorithm based on the proposed relational algebra for SPARQL [17] and semantics preserving SPARQL-to-SQL query translation [15]. This algorithm supports *Aggregate* queries and *Keyword Search* queries.
   - *Folder Node Construction and Querying.* We use the relational representation of triple RDF store, to store, manipulate, and query folder nodes.
   - *Path Node Construction and Querying.* To describe constraints on the path nodes, we reused expressions proposed in CSPARQL [4].

4. *Regular Expression Processor*: is responsible for parsing the described patterns through the nodes and edges in the graph. We developed a regular expression processor which supports optional elements (?), loops (+,*), alternation (—), and grouping ((...)).

5. *External Algorithm/Tool Controller*: is responsible for supporting applying external graph reachability algorithm or mining tools to the graph.

6. *Time-aware Controller*: RDF databases are not static and changes may apply to graph entities (i.e. nodes, edges, and folder/path nodes) over time. Time-aware controller is responsible for data changes and incremental graph loading. Moreover, it creates a monitoring code snippet and allocate it to a folder/path nodes in order to monitor its evolution and update its content over time. Details about timed folder/path nodes can be found in [11].

7. *Query Optimizer*: We described query optimizations for SPARQL and folder/path queries in [10]. In particular, to optimize the performance of queries, we developed four optimization techniques proposed in [14, 45, 15]: (i) selection of queries with specified varying degrees of structure and spanning keyword queries; (ii) selection of the smallest table to query based on the type information of an instance; (iii) elimination of redundancies in

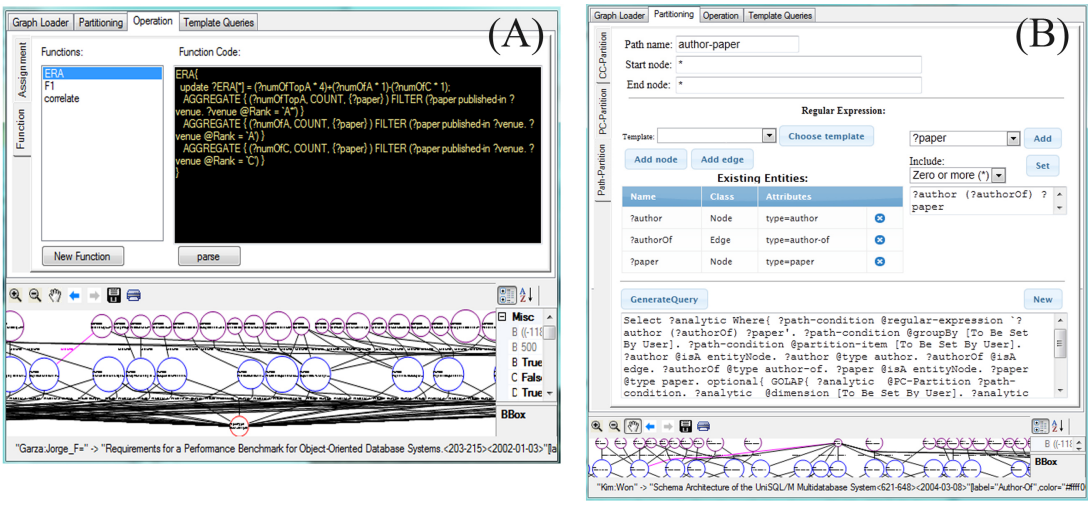


Figure 6.2: Screenshots of front-end tool for: (a) writing functions in Example 8; and (b) creating the regular expression and the path condition in Example 5.

basic graph pattern based on the semantics of the patterns and database schema; and (iv) create separate tables (property tables) for subjects that tend to have common properties to reduce the self-join problem. Moreover, we discussed optimization for GOLAP queries on graphs in Section 5.2.

8. *GOLAP controller*: is responsible for partitioning graphs (using folder and path nodes) and allows evaluation of OLAP operations on graphs independently for each partition, providing a natural parallelization of execution.

6.2 Implementation

We have implemented a graph processing engine, i.e. FPSPARQL, and details of our data model and query engine are presented in [10, 9]. In this work, we instrument the query engine with GOLAP (online analytical processing on graphs) controller, which is responsible for partitioning graphs (using folder and path nodes) and allows evaluation of OLAP operations on graphs independently for each partition, providing a natural parallelization of execution. Moreover, we provide a front-end tool for assisting users to create GOLAP queries in an easy way, and we update the physical storage layer to support better performance for GOLAP queries.

Front-End Tool. We have implemented a front-end tool for assisting users to create GOLAP queries, by providing an easy way to create partitions, selecting dimensions, and defining measures. To create CC-partitions, we provide list of graph objects (i.e. nodes and edges) and their attributes. To create PC/Path-Partitions, we provide an interface to easily create regular expressions to be used in path conditions. Figure 6.2-B illustrates how we created the regular expression and the path condition for the Example 5. We use frequent pattern discovery approaches to provide users with frequent patterns, which can be used to create regular expressions. We have created an interface to assist users writing and debugging operations. Figure 6.2-A illustrates how we use the tool to write functions for the Example 8. Finally, we provided users with a graph visualization tool for the exploration of partitions and query results (see Figures 6.2).

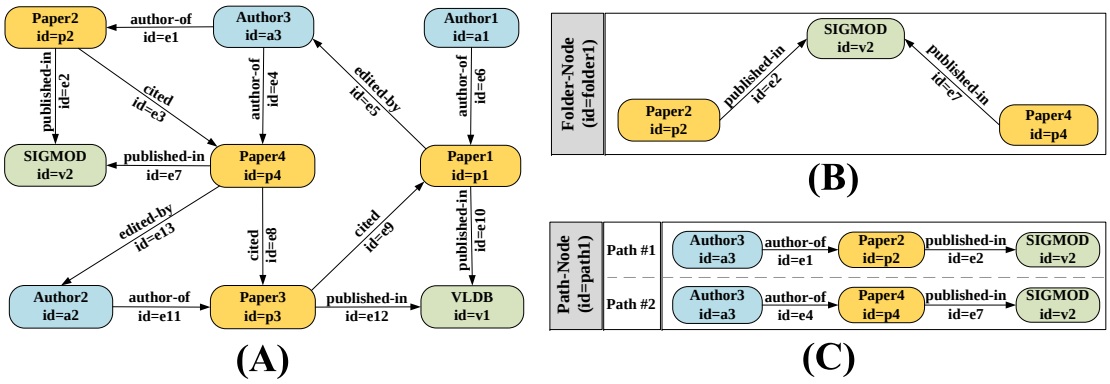


Figure 6.3: Representation of a sample: (a) DBLP graph; (b) folder node; and (c) path node.

Physical Storage Layer. The Resource Description Framework (RDF) represents a special kind of attributed graphs: in the RDF data model, graph edges can not be described by attributes. We model graphs based on a RDF data representation, where we support the uniform representation of nodes and edges in the graph. The simplest way to store a set of RDF statements is to use a relational database with a single table that includes columns for subject, property and object, i.e. triplestore. While simple, this schema quickly hits scalability limitations. To avoid this we developed a relational RDF store including its three classification approaches [45]: vertical (triple), property (n-ary), and horizontal (binary). Figure 6.4 illustrates the physical layer for storing the sample graph presented in Figure 6.3, where four types of objects (see Figure 6.4-A) can be stored: nodes, edges, folder-nodes, and path-nodes. Each object has the following mandatory attributes: ID (a unique identifier), class (can be set to entity-node, edge, folder, and path node), type (each node may conform to an entity type, e.g. author and paper), and label. Each object may have other user descriptive attributes, where these attributes can be defined in property stores.

As illustrated in Figure 6.4-B, we use triplestores to store each attribute. Each attribute may conform to an entity type, e.g. 'author:name' which means the attribute 'name' of type 'author'. We use the '@' symbol for representing attribute edges (e.g. the triple 'a1,@author:name,boualem' in Figure 6.3-B) and distinguishing them from the relationship edges (e.g. the triple 'a1,e1,p2' in Figure 6.3-C) between graph nodes. To store links between graph entities (i.e. nodes, folder-nodes, and path-nodes), we define triplestores for each edge type. For example, considering Figure 6.4-C, all the links typed as 'author-of' are stored in a separate triplestore. The link itself defined as an object and will be stored separately (see Figure 6.4-A).

In folder nodes, entities and relationships among them will be stored in folder-store (Figure 6.4-A). Folder node, as a graph entity, will have set of attributes. These attributes will be stored in folder-nodes-properties triplestores (see Figure 6.4-B). For example, consider the correlation condition *x.venue='SIGMOD'* where x is an instance of type *paper*. This query, groups set of papers published in 'SIGMOD' conference. As illustrates in the Figure 6.3-B the result of this query is the set {'paper2','paper4'}. We add a folder node to the original graph, and store the result of this query in the folder-store. Properties of this folder will be stored in folder-nodes-properties triplestores. In the folder store, the nodes under the column 'subject' (e.g. p_2 and p_4 in the folder-store represented in Figure 6.4-A) are the members of this folder.

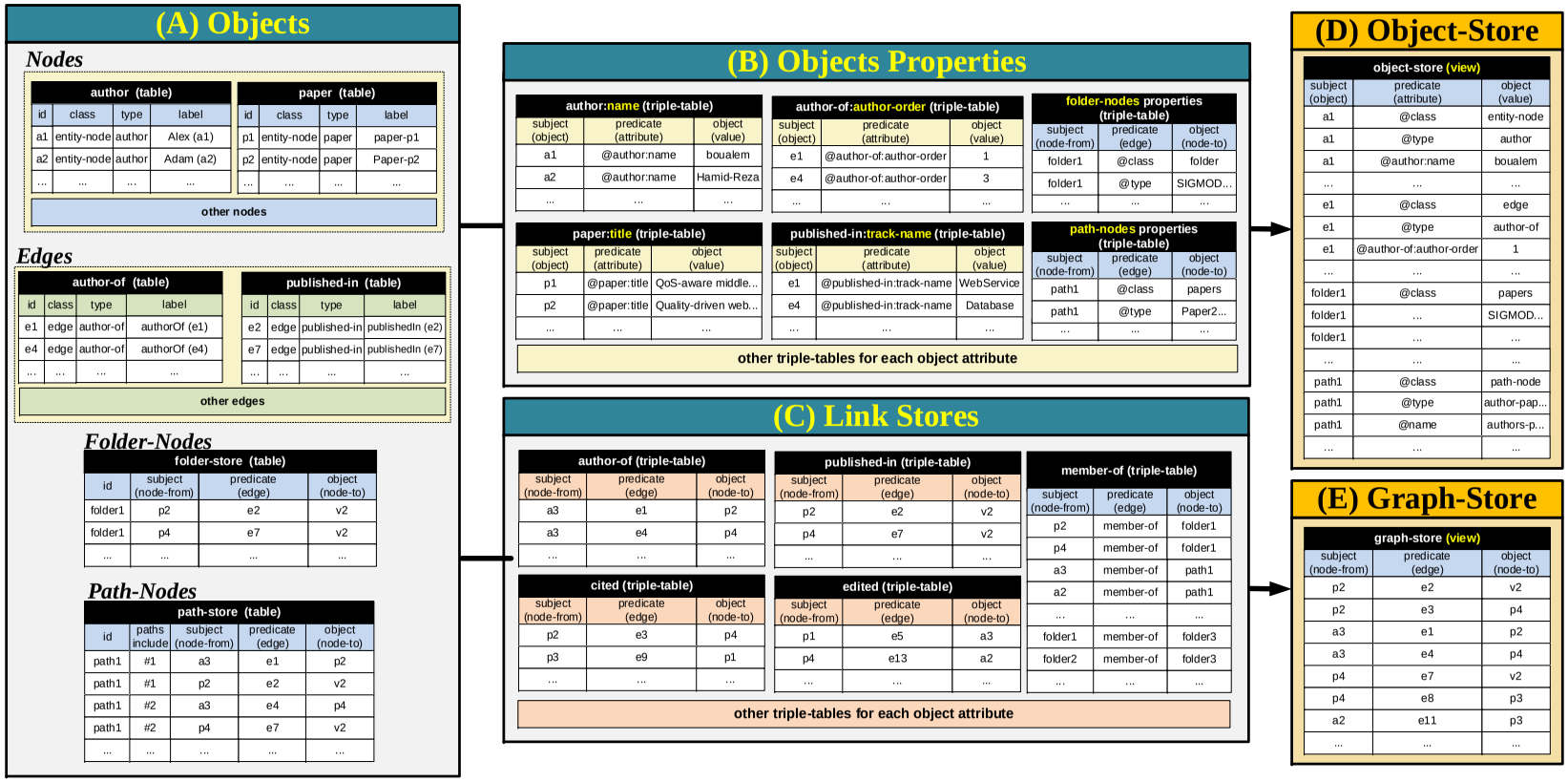


Figure 6.4: Physical layer for storing the sample graph represented in Figure 6.3 including: (A) object stores for storing nodes, edges, folder nodes, and path nodes; (B) object property store for storing objects attributes in triplestore format; (C) link stores for storing relationships among entities; (D) entity store as a view over object stores; and (E) graph store as a view over link stores.

A path node may contain set of related paths, where each path will be stored separately. A path node will be stored in path-store (Figure 6.4-A). A path node, may have set of attributes. These attributes will be stored in path-nodes-properties triplestores. For example, consider we are interested in finding occurrences of following pattern 'author3 (author-of) paper (published-in) venue2'. Set of paths can be discovered in the bibliographical network illustrated in Figure 6.3-A. The result of this query is represented in Figure 6.3-C. We add a path node to the original graph, and store the result of this query in the path-store. Properties of this path node will be stored in path-nodes-properties triplestores. In the path-store, each path (in a path node) will have a unique identifier, e.g., values '#1' and '#2' in the column 'paths-include' of table path-store (see Figure 6.4-A).

Finally, we create two views (i.e., object-store and graph-store) over the above explained physical layer. Object-Store (Figure 6.4-D) will include all the objects (i.e. nodes, edges, folder-nodes, and path-nodes) in the graph. Graph-Store (Figure 6.4-E) will include all the links among graph entities (i.e. nodes, folder-nodes, and path-nodes). Moreover, to optimize the performance of queries, we developed four optimization techniques proposed in [14, 45, 15]: (i) selection of queries with specified varying degrees of structure and spanning keyword queries; (ii) selection of the smallest table to query based on the type information of an instance; (iii) elimination of redundancies in basic graph pattern based on the semantics of the patterns and database schema; and (iv) create separate tables (property/link tables) for subjects that tend to have common properties to reduce the self-join problem.

6.3 Datasets

We carried out the experiments on two graph datasets:

- DBLP: This dataset was introduced in the example scenario. Recall from Section 3 that the DBLP graph we used in the experiments, contains over 1,670,000 nodes and over 2,810,000 edges.

- AMZLog (Amazon Online Rating System[1]): Amazon is one of the most popular online rating systems. One application of crowdsourcing[2] systems is evaluating and rating products on the web which refereed as Online Rating Systems. In an online rating system: (i) producers or sellers advertise their products; (ii) customers rate them based on their interests and experiences, the rating score is a number in the range; and (iii) the rating system aggregates scores received from customers to calculate a general rating score for every product. In this experiment, we use the rating log of Amazon online rating system that are collected by Leskovec et. al. [36] (i.e., referred in the following as AMZLog) for analyzing dynamics of viral Marketing by applying online analytical processing on this graph dataset. Online rating graph may contain some types of nodes (i.e. product, reviewer, category) and edges (e.g. hasRated in 'aleks hasRated book1'). Figure 6.5 illustrates a sample of data stored in AMZLog and its dataset statistics. We enriched Amazon graph by Adding attributes to nodes. For example, we enriched nodes typed as 'product' with (i) *rank in category*, to show how popular is the product in a particular category e.g. a book among all available books; and (ii) *overall rank*, to show the overall rank of a product in the system.

[1] http://snap.stanford.edu/data/amazon-meta.html

[2] Web 2.0 technologies have connected people all around the world. People can easily contact each other, collaborate on doing jobs or share information through mass collaborative systems which also are called crowdsourcing systems [18].

Dataset statistics	
Products	548,552
Product-Project Edges	1,788,725
Reviews	7,781,990
Product category memberships	2,509,699
Products by product group	
Books	393561
DVDs	19828
Music CDs	103144
Videos	26132

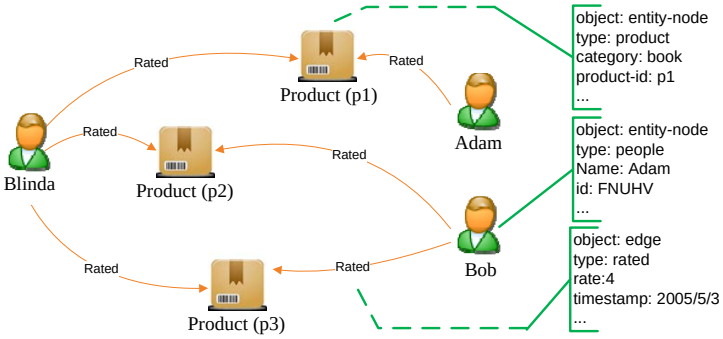


Figure 6.5: A sample of data stored in AMZLog.

We also enrich the 'reviewers' by adding: (i) *degree of interest*, to show in what extent a reviewer is interested in a particular category of products e.g. books, movies or albums; and (ii) *degree of expertise*, to show how dependable are the ratings that a reviewer has given to a particular product.

6.4 Evaluation

We report the evaluation results of the query engine extension, GOLAP, in terms of: (i) *performance*: we report performance in terms of running time of queries in seconds; (ii) *scalability*: we report scalability by characterizing how the performance of each query changes as the size of the graph increases; and (iii) *quality of results*: the quality of the results is assessed using classical *precision* metric which defined as the percentage of discovered results that are actually interesting. Moreover, we compare the performance of queries with and without query optimization techniques discussed in Section 5.2. Notice that, the performance and effectiveness of FPSPARQL query engine have been represented in [10]. Also in [12] we evaluated the performance of the FPSPARQL query engine compared to one of the well-known graph databases, the HyperGraphDB, which shows the great performance of the FPSPARQL query engine. All experiments were conducted on a HP PC with a 3.30 Ghz Core i5 processor, 8 GBytes of memory, and running a 64-bit Linux.

Performance. We report performance in terms of running time in seconds. We applied optimization techniques (see Section 5.2) on DBLP and AMZLog datasets. The optimization took 62 minutes for DBLP graph dataset and 41 minutes for AMZLog graph dataset. In the first step, we executed queries in Examples 4 to 8 on DBLP graph dataset. Figure 6.6-A illustrates the execution times for these queries. As illustrated in this figure, we divided each dataset into regular number of graph nodes (e.g. 0.1, 0.5, 0.9, 1.3, and 1.7 million nodes) and ran the experiment for different sizes of DBLP graph dataset. The evaluation shows a polynomial (nearly linear) increase in the execution time of the queries in respect with the dataset size. Recall from Section 5.2 that the partitions will be kept in memory and the operations will evaluated for one partition at a time.

In the second step, we provided 10 CC-Partition, 10 PC-Partition, and 10 Path-Partition queries (each having one operation) for DBLP dataset. We provided similar queries for AMZLog dataset. These queries were generated by domain experts who were familiar with the proposed datasets. Figures 6.6-B and C show the average execution time for applying the queries to DBLP

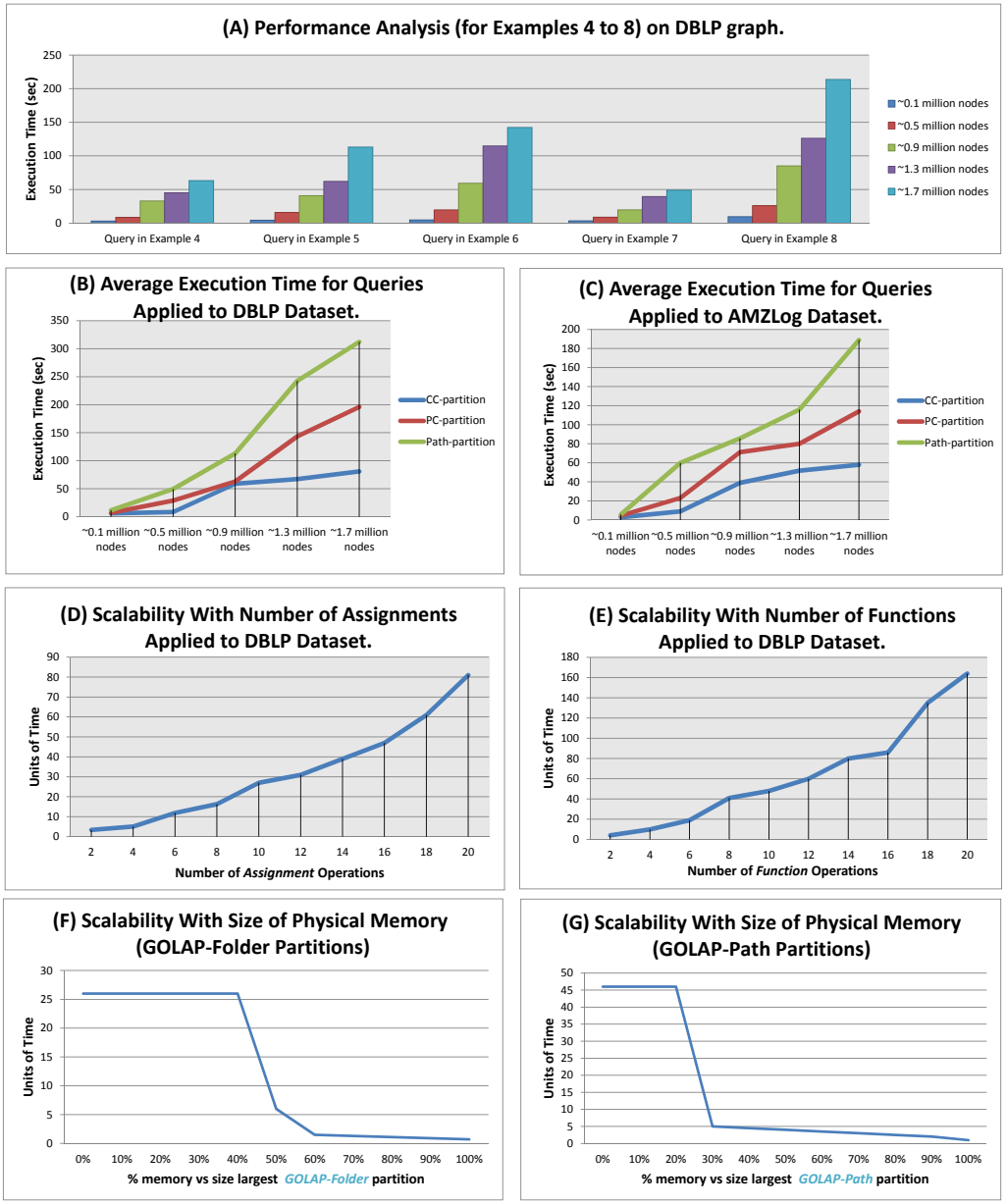


Figure 6.6: The evaluation results, illustrating: (A) performance analysis (for queries in Examples 4 to 8) applied on DBLP graph; (B) average execution time for 10 CC-Partition (blue line), 10 PC-Partition (red line), and 10 Path-Partition (green line) queries applied to different sizes of DBLP graph dataset; (C) average execution time for 10 CC-Partition (blue line), 10 PC-Partition (red line), and 10 Path-Partition (green line) queries applied to different sizes of AMZlog graph dataset; (D) scalability with number of *assignment* operations for 10 queries applied to DBLP dataset; (E) scalability with number of *function* operations for 10 queries applied to DBLP dataset; (F) scalability with size of physical memory for CC-Partitions and PC-Partitions; and (G) scalability with size of physical memory for Path-Partitions.

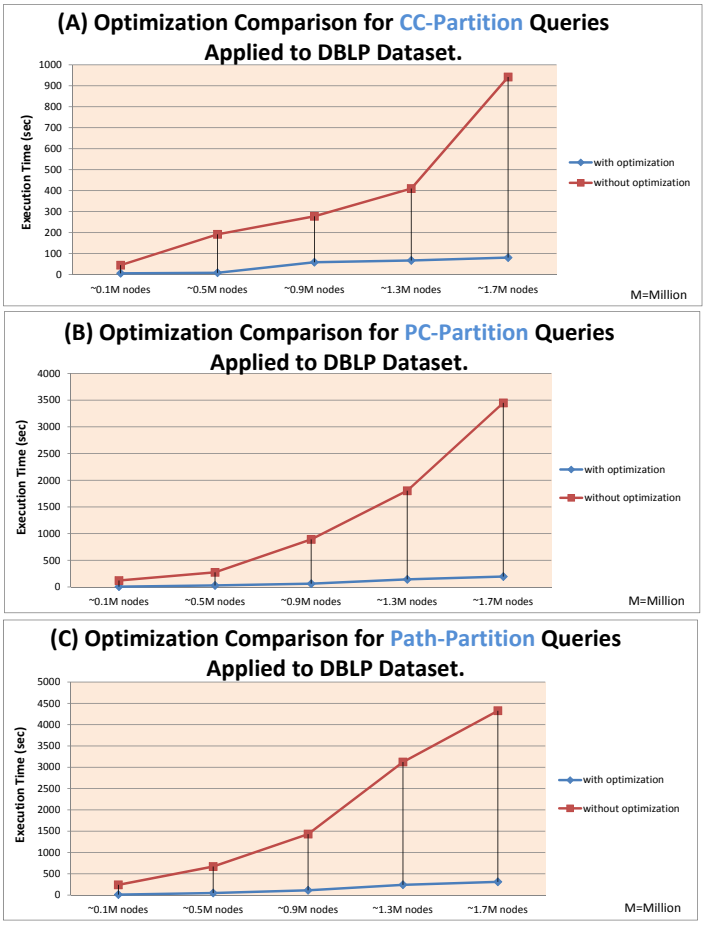


Figure 6.7: The query optimization results, illustrating: (A) Optimization Comparison for CC-Partition Queries Applied to DBLP Dataset; (B) Optimization Comparison for PC-Partition Queries Applied to DBLP Dataset; and (C) Optimization Comparison for Path-Partition Queries Applied to DBLP Dataset.

and AMZLog datasets respectively. As illustrated in this figure, we divided each log into regular number of graph nodes (same sizes for both DBLP and AMZlog graph) and ran the experiment for different sizes of graph datasets. In particular, the performance for applying queries on AMZlog is better, because in this dataset there is only 2 types of nodes and one type of relationships between them (see Figure 6.5). Consequently, the performance for path queries would be better, comparing DBLP dataset which contains four types of nodes and five types of relationships (see Figure 3.1). Finally, In Figures 6.7-A, 6.7-B, and 6.7-C we compare the performance of queries applied to DBLP dataset (i.e., CC/PC/Path-Partition queries in Figure 6.6-A) with and without query optimization.

Scalability. Another key measure of effectiveness of a query is its scalability. In our case, the key question to ask is how does each of the queries perform when the size of the graph increases? Figures 6.6-A,B, and C shows an almost linear scalability between the response time

of queries and the number of nodes in the graph. As another scalability metric we increased the number of *assignment* operations for 10 queries applied to DBLP dataset. Figure 6.6-D shows an almost linear scalability between the average response time of queries and the number of assignment operations. Moreover, we increased the number of *function* operations for 10 queries applied to DBLP dataset. Figure 6.6-E shows an almost linear scalability between the average response time of queries and the number of function operations.

Figures 6.6-F and G show the performance of our access structure as a function of available memory for folder-node partitions (CC-Partitions and PC-Partitions) and path-node partitions (Path-Partitions) respectively. The memory size is expressed as a percentage of the size required to fit the largest partition of data in the hash access structure in physical memory. Recall from Section 5.2 that for efficient access to single cells we built a partition level hash access structure and the partitions will be kept in memory and the operations will evaluated for one partition at a time. In the experiment for folder-node partitions, we execute a single assignment, $update?ar[?ap >= 200 AND ?ac >= 1000] = ?ar[?ap > 200 AND ?ac < 1000] * 1.5;$, from the query in Example 4. In the experiment for path-node partitions, we execute a single function, ERA, from the query in Example 8. In particular, if a partition does not fit in memory we incur an I/O if a referenced cell is not cached. In the case of folder-node partition (Figures 6.6-F), this occurs when the available memory is less than 40% of the largest partition, and for the path-node partition (Figures 6.6-G) this occurs when the available memory is less than 20% of the largest partition.

Quality. The quality of the results is assessed using classical *precision* metric which defined as the percentage of discovered results that are actually interesting. For evaluating the interestingness of the result, we asked domain experts who have the most accurate knowledge about the dataset to construct OLAP queries on graphs. For each query they codified their knowledge to use correlation/path conditions, construct regular expressions that describe paths through the nodes and edges in the graph, and describe dimensions and measures. They used the front-end tool (Figure 6.2) to construct GOLAP queries, visualize the content of constructed partitions, analyze discovered paths (using path conditions), and identify the query results to see what they consider relevant from an OLAP perspective. The quality evaluation applied on both DBLP and AMZLog datasets, where 18 queries constructed. For each dataset 9 queries constructed: three CC-Partition, three PC-Partition, and three Path-Partition queries, in which three queries applied to entity attributes, three queries applied to aggregated nodes, and three queries applied to inferred edges measures. As a result, all partitions and query results (e.g., updated/upserted measures) examined by domain experts and all considered relevant.

Discussion. We evaluated our approach using real-world datasets. As illustrated in Figure 6.6 we divide each dataset into regular number of nodes and ran the experiment for different sizes of datasets. The evaluation shows a polynomial (nearly linear) increase in the execution time of the queries in respect with the dataset size. Based on the lesson learned, we believe the quality of discovered paths is highly related to the regular expressions generated to find patterns in the log, i.e., generating regular expressions by domain experts will guarantee the quality of discovered patterns. Moreover, the performance of partitioning the graph using path-conditions (e.g., in PC-Partitions and Path-Partitions) is highly related to the reachability algorithms used for discovering paths among entities.

We developed an interface to support various graph reachability algorithms [2] such as Transitive Closure, GRIPP, Tree Cover, Chain Cover, Path-Tree Cover, and Shortest-Paths [26]. In general, there are two types of graph reachability algorithms [2]: (1) algorithms traversing from starting vertex to ending vertex using breadth-first or depth-first search over the graph, and

(2) algorithms checking whether the connection between two nodes exists in the edge transitive closure of the graph. Considering $G = (V, E)$ as directed graph that has n nodes and m edges, the first approach incurs high cost as $O(n+m)$ time which requires too much time in querying. The second approach results in high storage consumption in $O(n^2)$ which requires too much space. In this experiment, we used the GRIPP [53] algorithm which has the querying time complexity of $O(m - n)$, index construction time complexity of $O(n + m)$, and index size complexity of $O(n + m)$. Moreover, as discussed in Section 5.2, we used a path-based indexing [56] (and cycle elimination [25, 59]) technique to build path indices and partitioning the graph based on patterns among graph entities. Overall, the evaluation shows that the approach is performing well.

7 Conclusion and Future Work

In this paper, we have proposed a graph data model, GOLAP, and a query language for online analytical processing on graphs. We use folder and path nodes to support multi-dimensional and multi-level views over large graphs. We extended FPSPARQL to support n-dimensional computations. We described optimizations and execution strategies possible with the proposed extensions. We proposed two types of OLAP operations on graphs, assignments and functions. Assignments defined to apply operations on entity attributes. Functions defined to apply set of related operations on network structures among entities. Operations support UPSERT and UPDATE semantics. We have conducted experiments over real-world datasets and the evaluation shows the viability and efficiency of our approach. A front-end tool has been provided to assist users with generating GOLAP queries in an easy way.

As future work, we plan to design a visual query interface to support users in expressing their queries over the conceptual representation of the GOLAP graph in an easy way. We plan to employ interactive graph exploration and visualization techniques (e.g. storytelling systems [46, 47]) to help users quickly identify the interesting parts of the graph. Moreover, we are performing an empirical study of computations on very large graphs (e.g., the social graph of Facebook contains more than 700 million active users) in three well-studied platform models: a relational model, a data-parallel model (support MapReduce-like abstractions), and a special-purpose in-memory model. Finally, we plan to present a GOLAP data service, i.e., a combination of the runtime and a web service through which the services are exposed.

Bibliography

[1] Alberto Abelló and Oscar Romero. On-line analytical processing. In *Encyclopedia of Database Systems*, pages 1949–1954. Springer US, 2009.

[2] Charu C. Aggarwal and Haixun Wang. *Managing and Mining Graph Data*. Springer Publishing Company, Incorporated, 2010.

[3] Fuat Akal, Klemens Böhm, and Hans-Jörg Schek. Olap query evaluation in a database cluster: A performance study on intra-query parallelism. In *ADBIS*, pages 218–231, 2002.

[4] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending sparql with regular expression patterns. *J. Web Sem.*, 7(2):57–73, 2009.

[5] Kemafor Anyanwu, Angela Maduko, and Amit Sheth. Sparq2l: towards support for subgraph extraction queries in rdf databases. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 797–806, New York, NY, USA, 2007. ACM.

[6] Andrey Balmin, Thanos Papadimitriou, and Yannis Papakonstantinou. Hypothetical queries in an olap environment. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 220–231. Morgan Kaufmann, 2000.

[7] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In *WWW*, pages 1061–1062, 2009.

[8] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. Codebook: discovering and exploiting relationships in software repositories. ICSE'10, pages 125–134, 2010.

[9] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, and Hamid Reza Motahari-Nezhad. An artifact-centric activity model for analyzing knowledge intensive processes. Technical report: 201210, University of New South Wales, 2012.

[10] Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Hamid Reza Motahari-Nezhad, and Sherif Sakr. A query language for analyzing business processes execution. 9th International Conference on Business Process Management (BPM), pages 281–297. Springer-Verlag Berlin Heidelberg, 2011.

[11] Seyed-Mehdi-Reza Beheshti, Hamid Reza Motahari-Nezhad, and Boualem Benatallah. Temporal Provenance Model (TPM): Model and query language. Technical report: 1116, University of New South Wales, 2010.

[12] Seyed-Mehdi-Reza Beheshti, Sherif Sakr, Boualem Benatallah, and Hamid Reza Motahari-Nezhad. Extending SPARQL to support entity grouping and path queries. Technical report, unsw-cse-tr-1019, University of New South Wales, 2010.

[13] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[14] Artem Chebotko, Shiyong Lu, Xubo Fei, and Farshad Fotouhi. Rdfprov: A relational rdf store for querying and managing scientific workflow provenance. *Data Knowl. Eng.*, 69(8):836–865, 2010.

[15] Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. Semantics preserving sparql-to-sql translation. *Data Knowl. Eng.*, 68(10):973–1000, 2009.

[16] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and Philip S. Yu. Graph olap: Towards online analytical processing on graphs. In *ICDM*, pages 103–112, 2008.

[17] Richard Cyganiak. A relational algebra for SPARQL. 2005.

[18] AnHai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96, 2011.

[19] Anton Dries, Siegfried Nijssen, and Luc De Raedt. A query language for analyzing networks. In *CIKM*, pages 485–494, 2009.

[20] Leo Egghe. Theory and practise of the g-index. *Scientometrics*, 69(1):131–152, 2006.

[21] Lorena Etcheverry and Alejandro A. Vaisman. Enhancing olap analysis with web cubes. In *ESWC*, pages 469–483, 2012.

[22] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 175–184, New York, NY, USA, 2010. ACM.

[23] Camille Furtado, Alexandre A. B. Lima, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. Physical and virtual partitioning in olap database clusters. In *SBAC-PAD*, pages 143–150, 2005.

[24] Leticia I. Gómez, Silvia A. Gómez, and Alejandro A. Vaisman. A generic data model and query language for spatiotemporal olap cube analysis. In *EDBT*, pages 300–311, 2012.

[25] Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 499–508, New York, NY, USA, 2010. ACM.

[26] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Fast and accurate estimation of shortest paths in large graphs. CIKM'10, pages 499–508, 2010.

[27] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *SIGMOD Conference*, pages 1–12, 2001.

[28] Jiawei Han, Yizhou Sun, Xifeng Yan, and Philip S. Yu. Mining knowledge from data: An information network analysis approach. In *ICDE*, 2012.

[29] Jiawei Han, Xifeng Yan, and Philip S. Yu. Scalable olap and mining of information networks. In *EDBT*, 2009.

[30] Jorge E. Hirsch. An index to quantify an individual's scientific research output that takes into account the effect of multiple coauthorship. *Scientometrics*, 85(3):741–754, 2010.

[31] David A. Holl, Uri Braun, Diana Maclean, Kiran kumar Muniswamy-reddy, and Margo I. Seltzer. Choosing a data model and query language for provenance, 2008.

[32] Ming Ji, Yizhou Sun, Marina Danilevsky, Jiawei Han, and Jing Gao. Graph regularized transductive classification on heterogeneous information networks. In *ECML/PKDD (1)*, pages 570–586, 2010.

[33] Benedikt Kämpgen and Andreas Harth. Transforming statistical linked data for use in olap systems. In *I-SEMANTICS*, pages 33–40, 2011.

[34] Benedikt Kmpgen, Sean O'Riain, and Andreas Harth. Interacting with statistical linked data via olap operations. In *International Workshop on Interacting with Linked Data (ILD 2012), Extended Semantic Web Conference (ESWC)*, 2012.

[35] Krys J. Kochut and Maciej Janik. Sparqler: Extended sparql for semantic association discovery. ESWC'07, pages 145–159, Berlin, Heidelberg, 2007. Springer.

[36] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The dynamics of viral marketing. *TWEB*, 1(1), 2007.

[37] Alexandre A. B. Lima, Marta Mattoso, and Patrick Valduriez. Adaptive virtual partitioning for olap query processing in a database cluster. *JIDM*, 1(1):75–88, 2010.

[38] H.R. Motahari-Nezhad, R. Saint-Paul, F. Casati, and B. Benatallah. Event correlation for process discovery from web service interaction logs. *VLDB J.*, 20(3):417–444, 2011.

[39] Thomas Neumann and Gerhard Weikum. Rdf3x: a riscstyle engine for rdf. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, pages 647–659, 2008.

[40] Beng Chin Ooi, Bei Yu, and Guoliang Li. One table stores all: Enabling painless free-and-easy data publishing and sharing. CIDR'07, pages 142–153, 2007.

[41] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf (working draft). Technical report, W3C, March 2007.

[42] Tieyun Qian, Yang Yang, and Shuo Wang. Refining graph partitioning for social network clustering. In *WISE*, pages 77–90, 2010.

[43] Qiang Qu, Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hongyan Li. Efficient topological olap on information networks. In *DASFAA*, pages 389–403, 2011.

[44] Oscar Romero and Alberto Abelló. A survey of multidimensional modeling methodologies. *IJDWM*, 5(2):1–23, 2009.

[45] Sherif Sakr and Ghazi Al-Naymat. Relational processing of rdf queries: a survey. *SIGMOD Rec.*, 38(4):23–28, 2009.

[46] Arjun Satish, Ramesh Jain, and Amarnath Gupta. Tolkien: an event based storytelling system. *Proc. VLDB Endow.*, 2:1630–1633, August 2009.

[47] Ariel Shamir and Alla Stolpnik. Interactive visual queries for multivariate graphs exploration. *Computers & Graphics*, 36(4):257–264, 2012.

[48] Yizhou Sun, Charu C. Aggarwal, and Jiawei Han. Relation strength-aware clustering of heterogeneous information networks with incomplete attributes. *PVLDB*, 5(5):394–405, 2012.

[49] Yizhou Sun, Jiawei Han, Peixiang Zhao, Zhijun Yin, Hong Cheng, and Tianyi Wu. Rankclus: integrating clustering with ranking for heterogeneous information network analysis. In *EDBT*, pages 565–576, 2009.

[50] Yizhou Sun, Yintao Yu, and Jiawei Han. Ranking-based clustering of heterogeneous information networks with star network schema. In *KDD*, pages 797–806, 2009.

[51] Erik Thomsen. *Olap Solutions: Building Multidimensional Information Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[52] Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. In *SIGMOD Conference*, pages 567–580, 2008.

[53] Silke TriBl and Ulf Leser. Fast and practical indexing and querying of very large graphs. SIGMOD '07, pages 845–856, NY, USA, 2007. ACM.

[54] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Gregory Dorman, Nathan Folkert, Abhinav Gupta, Lei Sheng, and Sankar Subramanian. Spreadsheets in rdbms for olap. In *SIGMOD Conference*, pages 52–63, 2003.

[55] Dong Xin, Zheng Shao, Jiawei Han, and Hongyan Liu. C-cubing: Efficient computation of closed cubes by aggregation-based checking. In *ICDE*, page 4, 2006.

[56] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *SIGMOD Conference*, pages 335–346, 2004.

[57] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[58] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and olap multidimensional networks. In *SIGMOD Conference*, pages 853–864, 2011.

[59] Lei Zou, Peng Peng, and Dongyan Zhao. Top-k possible shortest path query over a large uncertain graph. In *WISE*, pages 72–86, 2011.