Alma Mater Studiorum – Università di Bologna

DOTTORATO DI RICERCA IN

Computer Science and Engineering

Ciclo XXX

Settore Concorsuale: 01/B1 Informatica

Settore Scientifico Disciplinare: INF/01 Informatica

A new Nested Graph Model for Data Integration

Presentata da: Giacomo Bergami

Coordinatore Dottorato	**Supervisore**
Prof. Paolo Ciacccia	**Prof. Danilo Montesi**

Esame finale anno 2018

A new Nested Graph Model for Data Integration

Giacomo Bergami

Dottorato di Ricerca in Computer Science and Engineeering
Ciclo XXX
Settore concorsuale di afferenza: 01/B1 - Informatica
Settore scientifico disciplinare: INF/01 - Informatica

Coordinatore:	Supervisore:
Paolo Ciaccia	Danilo Montesi

Alma Mater Studiorum - Università di Bologna   Esame finale anno 2018

Contents

> *"No plan can predict everything. Some people will raise their heads, others will mutiny. The time will not cease to bestow losses and fame to whose who will continue the fight. [. . .] Do not pursue your actions according to a plan."*
>
> — Luther Blissett, *Q*, Epilogue

Abstract

Despite graph data gained increasing interest in several fields, no data model suitable for both querying and integrating differently structured graph and (semi)structured data has been currently conceived. The lack of operators allowing combinations of (multiple) graphs in current graph query languages (graph joins), and on graph data structure allowing neither data integration nor nested multidimensional representations (graph nesting) are a possible motivation. In order to make such data integration possible, this thesis proposes a novel model (General Semistructured data Model) allowing the representation of both graphs and arbitrarily nested contents (e.g., one node can be contained by more than just one parent node), thus allowing the definition of a nested graph model, where both vertices and edges may include (overlapping) graphs.

We provide two graph joins algorithms (Graph Conjunctive Equijoin Algorithm and Graph Conjunctive Less-equal Algorithm) and one graph nesting algorithm (Two HOp Separated Patterns). Their evaluation on top of our secondary memory representation showed the inefficiency of existing query languages' query plan on top of their respective data models (relational, graph and document-oriented). In all three algorithms, the enhancement was possible by using an adjacency list graph representation, thus reducing the cost of joining the vertices with their respective outgoing (or ingoing) edges, and by associating hash values to both vertices and edges.

As a secondary outcome of this thesis, a general data integration scenario is provided where both graph data and other semistructured and structured data could be represented and integrated into the General Semistructured data Model. A new query language outlines the feasibility of this approach (General Semistructured Query Language) over the former data model, also allowing to express both graph joins and graph nestings. This language is also capable of representing both traversal and data manipulation operators.

1 Introduction

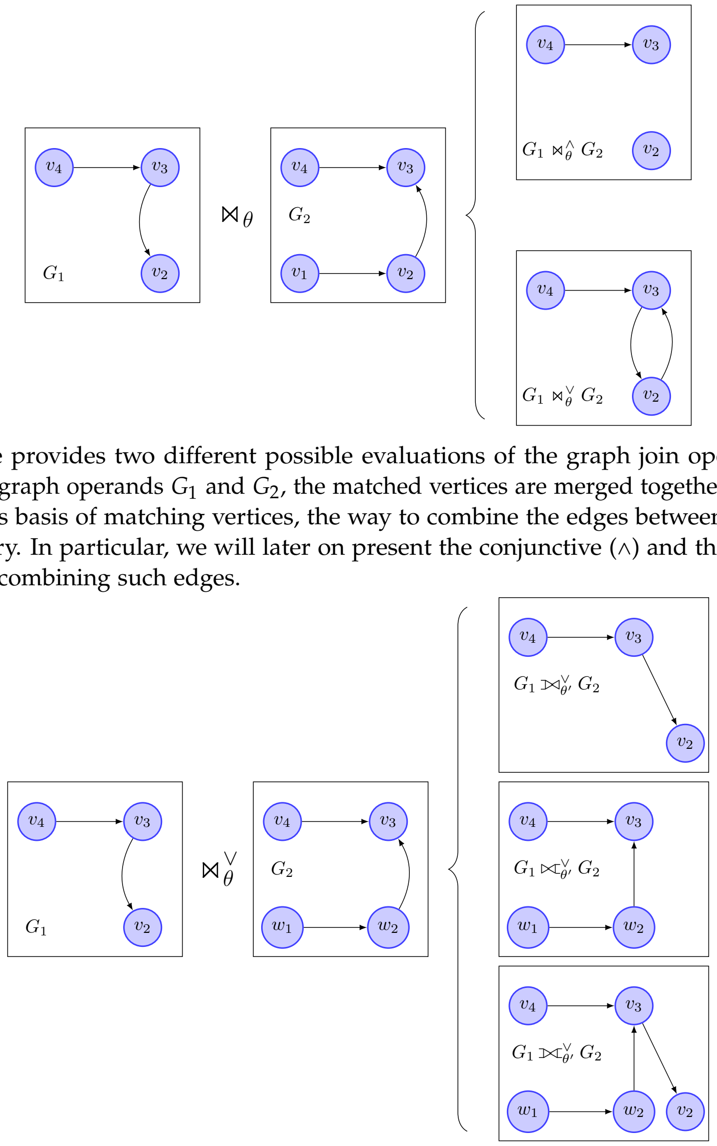
Contents

Despite that modern Graph Database Management Systems (GDBMSs) are a well-known subject of studies, three fundamental aspects are missing in literature: operators integrating and combining graphs, the adoption of such operators in current graph query languages, and the definition of a graph data structure providing both a multidimensional and nested representation. As regards of the aforementioned graph integration operators, both graph joins generalizing the graph products and graph nestings summarizing graph data with other extracted graph patterns are required.

In particular, this thesis is going to focus on two graph operators allowing to combine graphs: **graph joins** and **graph nestings**. First, the graph join operator (Chapter 4 on page 89) is useful in practical scenarios, such as the intersection or the merge of different transport overlay networks, or for comparing and merging different ontology representations. The flexibility at the edge combination level provides adaptiveness to such operator (Figure 1.1a). Moreover, if we fix a same theta-join predicate and a specific edge semantics, we could further on extend the definition of the graph join as in the relational model, that is by allowing that even the non matching vertices from one (left and right join) or both operands (full join) must also appear (Figure 1.1b). Hereby, the graph join operator defines a whole class of graph operators (**graph $\otimes_\theta$-products**), that can be differently instantiated as required by the user, and hence can be also differently implemented.

Graph nesting (Chapter 7 on page 195) permits the summarization of graph data within one graph data source: in particular, we could "fuse" or "summarize" all the vertices belonging to the same cluster, and then we could group the paths occurring between the fused vertices as edges containing such paths. As a consequence, such operator provides dimensionality reduction that is often required in multidimensional data analysis. Group Recommendation Systems characterize a use case scenario in which the members sharing similar interests and their suggested activities could be "fused", thus providing a coarser representation of the sequence of similar activities. While the class of graph join operators can be defined on top of current graph representations, graph nesting requires an extension of the graph data model for fulfilling the aforementioned graph data integration purposes. Such extension must allow to represent graphs within both vertices and edges (nestings), thus representing a summarized view of the contained graph. Nested representations are shared among task representations and complex world knowledge representations, such as medical procedures for specific diseases such as glaucoma [KW82].

Graphs distinguish two different classes of data types, vertices and edges. Edges are constrained by the ids of the vertices through source and destination dependencies. In particular, most graph operators require the execution of distinct operations over the vertex and edge sets. Nevertheless, current graph data structures do not allow a direct data integration with semistructured data [Rol13, PSAH16] (and hence, even structured data) because graph definitions such as property graphs and RDFs do not permit nested representations, neither in vertices nor in edges. *Expert systems* are an example of practical use cases requiring the features in the previous generalized data integrations; such systems could be either general purpose (such as *IBM Watson* [oRD12] and *DeepDive* [PAKR16]) or more domain knowledge



(a) This picture provides two different possible evaluations of the graph join operators: in both cases, given the graph operands G_1 and G_2, the matched vertices are merged together into one single vertex. Over this basis of matching vertices, the way to combine the edges between these matched vertices may vary. In particular, we will later on present the conjunctive ($\wedge$) and the disjunctive ($\vee$) "semantics" for combining such edges.

(b) Given two graphs G_1 and G_2, we provide the result of three possible outer joins over the disjunctive graph join: vertices sharing the same id identify vertices with the same value. These figure make explicit that the left/right full outer interpretation returns both the vertices that are shared among the graphs and the vertices belonging to the left/right graph.

Figure 1.1 Introducing Graph Joins over standard graphs.

related, such as healthcare settings (medical diagnosis formulation [KW82] and health condition prediction methods [BBMP15]). For this reason the present thesis proposes a new graph data model, named **nested graphs**, allowing the embedding (or *nesting*) of both vertices and edges for both data class types, hence permitting graph nesting operations. Nested graphs allow to directly represent semistructured documents. In particular, such data model relies on a broader semistructured model, General Semistructured Model (**GSM**), which both overcomes current data models' limitations and is proposed by this thesis.

Last, we must ask ourselves which is a proper language to query nested data structures, including nested graphs: given the aforementioned considerations, such query language should be able to express queries for both (semi)structured and graph data. In order to complete the data integration of (semi)structured with graph data, such language should express data transformations towards the most general data representation - our nested graphs - and to uniform the sources' data schema in one single user-provided final schema. In order to meet such research goals, this thesis proposes the **GSQL** query language over GSMs, thus allowing to formally characterize the Global as A View data integration scenario in its entirety. The graph join and nesting operator, the definition of a graph query language for data integration and the outline of a data integration system for any sort of data representation are all required features missing in current literature, which are provided by the current thesis. In particular, the adoption of these solutions allows to achieve these further outcomes:

- A new property graph representation allowing to optimize the graph join operators by exploiting primary and secondary indices. This data structure also allows an easy parallelization by splitting the data in different vectors. This data structure is extended to efficiently support a nested data representation.
- Current query languages optimization strategies do not efficiently implement graph joins and nestings. The definition of graph joins in current query languages is verbose, while the proposed graph join definition provides a more user-friendly representation.
- The definition of GSQL (Chapter 6 on page 157) allows the expression within the same syntax of both graph and semistructured pattern query languages, alongside with set, relational and semistructured operations. On the other hand current graph query languages often provides these features singularly at an higher abstraction level.
- GSQL query language allows the implementation of all the data integration operators required in the Global As A View approach.

1.1 Graph Data: Use Cases

Graph databases are already used in tasks requiring the integration of both schemaless (or time-variant schema) data alongside with structured information [PJMR14, SAZ11]. Graphs are used as an intermediate representation during the data transformation phase, thus allowing to integrate intermediate graph representation with other graph data. In addition to Social Networks [DMR16], graph data is also used in the following scenarios:

Analysing (Hyper)texts. Even if hypertext contents are usually provided as semi-structured data using markup languages like XML, some recent work proposed alternative graph representations.

Firstly, if we want to focus on the mere syntactic representation of semi-structured data, we could analyse how the author used the XML tags to produce the document, and investigate which structural patterns were used. In this case an RDF representation [LS99, GHMP11] provides a ideal schema-independent data [IPPV14, BDIP+13] that could be used in automatic reasoners such as **Jena** [CDD+04] and **Pellet** [SPG+07] to extract structural informations.

Finally, graphs can provide a semantic interpretation of the textual content [VnI11]. Such representation allows to later perform either specific graph mining algorithms [SHJ+13] such as clustering and association rules, or basic graph metric operations such as betweenness centrality and degree distribution [New10]. In this context, graphs are also used to represent knowledge bases such as **BabelNet** or **WordNet**, for multi-word recognition [LVJRT14], word similarity [HRJM15] and multilingual word disambiguation [NP12b].

Temporal Data. One of the problems[1] of the relational model is its inability to represent temporal data. Many possible representations for temporal graphs structures have been proposed. The most simple data model is to represent the evolution of a network through time via distinct graph snapshots [Khu12]. This multi-layer network model is not very informative since no information is provided to understand how the different layers are related to each other.

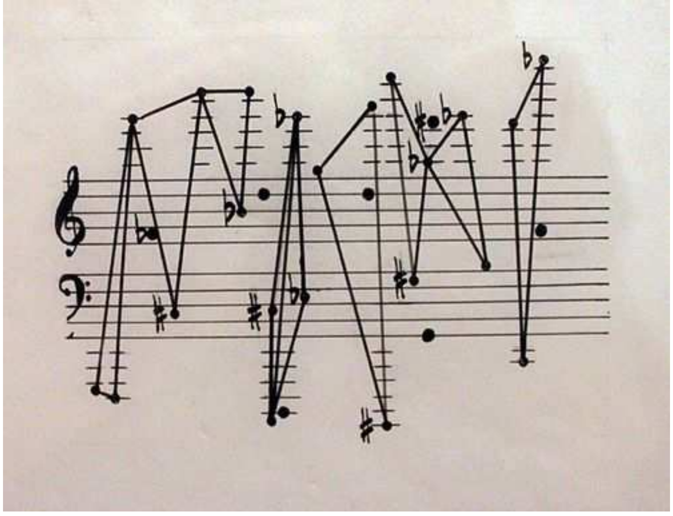
Temporal information can be represented as a single graph [WCH+14]: each vertex represents an agent and each edge expresses an interaction, which time span is represented through the edge's weight. Such representation suffers from interpretation problems: firstly, when cyclic interactions occur it is not clear which node started the communication process, and secondly, nodes having different interaction times are not taken into account. All those issues are solved by another multi-layer graph representation [Kos08], where each layer describes the snapshot of the interactions occurring at a given time, while the extra-layer edges link the same agents interacting subsequent time steps.

MOLAP representation. Business Intelligence applications may also use graph databases for MOLAP multi-dimensional data warehouses, in which different types of data are integrated [PJMR14, SAZ11]. Given that vertices can be associated to distinct dimensions [PMB+17], graphs allow data warehouse multidimensional queries [CYZ+08, ZLXH11, EV12a]. Companies are already generating graph data as Enterprise Social Networks, which are used as an infrastructure for internal communication and opinion mining. Graphs allow the measurement of enterprise success [RHKB13] and to design company workflows [PAK16]. Such enterprise graph data could be integrated with external business-oriented OnLine Social Networks (e.g. LinkedIn, Ask-a-peer).

[1] See Section 3.1.2 on page 58 for further problems related to the relational model.

Part I

Related Works



John Cage, Concert for Piano and Orchestra *Page 18 of Solo for Piano*

An example of unstructured information in contemporary music. The graph's vertices are musical notes, while the edges' interpretation may be freely interpreted by the player.

2 Data integration: a data representation-independent approach

Contents

> *Logic is the most useful tool of all the arts. Without it no science can be fully known. It is not worn out by repeated use, after the manner of material tools, but rather admits of continual growth through the diligent exercise of any other science. For just as a mechanic who lacks a complete knowledge of his tool gains a fuller [knowledge] by using it, so one who is educated in the firm principles of logic, while he painstakingly devotes his labor to the other sciences, acquires at the same time a greater skill at this art.*
>
> — William of Ockham, *Summa Logicæ*, Prefatory Letter

Data Integration is a sequence of transformations (and hence, queries) through which all the data coming from different data sources are mapped (**transcoded**) to a *reconciled representation* over a many-to-one data association (**alignment**). Each resulting value-representation (type) may be indistinguishable from the representation available from the data sources (uniform representation).

Data integration theory uses abstraction to generalise all the possible approaches which are data structure specific. Such theory focuses on the main preprocessing steps which are alignment and transcoding as introduced in the incoming introductory examples (Section 2.1). While languages and models outlining data representation abstraction were defined decades ago and date back to the studies on programming languages [omg96, Pie02], the theoretical approaches abstracting data integration process are more recent [Len02, DGLL+17]. This lack of formalisation made all the researchers focus more on the data integration aspect that are representation-dependent (*how shall we integrate data with different "shapes"*) than on the actual integration process (*what makes data from different* *sources "hard" to integrate, independently from their representation?*). The latter takes stock of two different integrations (or fusion[1]) which are "In-Database integration" (Section 2.2.2) and "Multi-database integration" (Section 2.3). These specific topics allow to draw the following observations for both current query languages and data structures:

- *In-Database Integration* requires that joins and group by operations are generalised to the nesting operator, which enables structural aggregations (Section 2.2.2). Structural aggregations should allow representing data at different abstraction levels, where summarised and coarser representations coexist (Example 8 on page 44). Current query languages and data structures do not meet these requirements.
- *Multi-database integration* requires that a query language should support all the operators used on different data representations (Section 2.3.3) alongside with transcoding (Section 2.3.1). Data representation should be able to represent data and alignments uniformly.

2.1 Preliminaries: data representation dependent approach

The data integration approaches over specific data representations produced a vast amount of literature with little scientific advancement. The lack of generality forced the authors to repeat the same strategies under different possible data structures (between *semistructured* XML data [Pog06], among *relational tables* [MM09], between *structured* and *semistructured* documents [MFK01, Lu06, MM06]). Moreover, the lack of uniformity in these approaches led to similar results achieved at different abstraction levels: compare [GMP+12] with the general schema alignment approach in [ES13]. This latter approach proved to be interesting on the long run: it is recently used both theoretical [VMT15] and more practical scenarios, such as data integration among federated data warehouses with different schemas [GMP+12].

Instead of associating an uncertainty to each schema representation and providing all the possible combinations for schema integrations [MM09], from the very beginning of this thesis, we're going to consider a more flexible approach considering only the final schema to which all the data sources must comply.

Even though we're going to discuss the difference between structured and semistructured data in depth in Chapter 3, we're going to show how the approaches as mentioned above could be all generalised using the schema alignment (also known as "ontology alignment") approach. We refer to such chapter for additional details concerning the data representations that are here just mentioned in passing. We provide some examples on how to integrate different semistructured formats (representing graphs) into graphs (Section 5.4.2 on page 143) and vice versa (Section 6.3.3 on page 182) in the following chapters of this thesis.

[1] In current literature, the term "fusion" assumes an ambiguous connotation: while anglophone literature [HL97, KKKR13] focuses on the integration of sensor unstructured data possibly with structured data (e.g. geospatial information), germanophone literature [LN07, BN09] calls "data fusion" (*Datenfusion*) the process of data cleaning and conflict resolution that could be carried out within one same data source. This terminological ambiguity reveals two different aspects of performing data integration.

SocialSecurityNo	diagnosis	week	month	year	ward	ICD-9-CM
BCDVHZ59S23F743S	Right parotid neoplastic formation	1	1	2017	Oncology	210.2
PNPMZZ74H45H782P	Relapsing epistaxis	2	1	2017	Emergency	784.7
PKTBMF36E14H842O	Septal deviation and nasal-sinus polyps	3	1	2017	Emergency	748.1, 471.0

(a) `Admissions` table from the internal Data Warehouse.

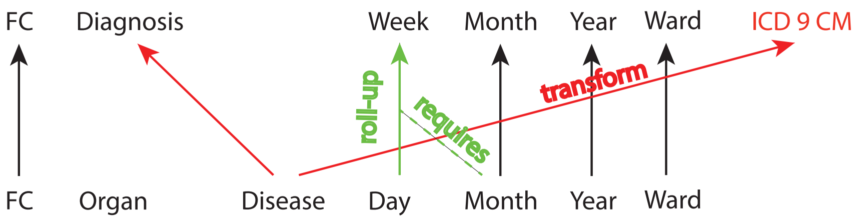
SocialSecurityNo	organ	disease	day	month	year	ward
BCDVHZ59S23F743S	`NULL`	Right vestibular deficit	2	1	2017	Emergency
PNPMZZ74H45H782P	Appendix	Severe Appendicitis	17	2	2017	Emergency
PKTBMF36E14H842O	Intestine	Recanalization of Crohn's Disease	25	3	2017	Emergency

(b) `Hospitalization` table from an external Data Warehouse.

SocialSecurityNo	diagnosis	week	month	year	ward	ICD-9-CM
BCDVHZ59S23F743S	Right parotid neoplastic formation	1	1	2017	Oncology	210.2
BCDVHZ59S23F743S	Right vestibular deficit	2	1	2017	Emergency	386.10
PNPMZZ74H45H782P	Relapsing epistaxis	2	1	2017	Emergency	784.7
PNPMZZ74H45H782P	Severe Appendicitis	8	2	2017	Emergency	540.9
PKTBMF36E14H842O	Septal deviation and nasal-sinus polyps	3	1	2017	Emergency	748.1, 471.0
PKTBMF36E14H842O	Recanalization of Crohn's Disease	15	3	2017	Emergency	555.0

(c) Expected result for the integration of the two upper tables.

Figure 2.1 Integrating two tables (a and b) pertaining to hospitalization into one final table (c), matching with the schema of b. Diagnoses and diseases were obtained from the "ICD code it" dataset, available at `smartdata.cs.unibo.it`, while the other parts are randomly generated.



(a) Alignment between the schemas of the two data sources. Data types associated to each field are not showed.

SocialSecurityNo	disease	week	month	year	ward	ICD-9-CM
BCDVHZ59S23F743S	Right vestibular deficit	$\varsigma(2,1)$	1	2017	Emergency	ς'(Rigth. . .)
PNPMZZ74H45H782P	Severe Appendicitis	$\varsigma(17,2)$	2	2017	Emergency	ς'(Severe. . .)
PKTBMF36E14H842O	Recanalization of Crohn's Disease	$\varsigma(25,1)$	3	2017	Emergency	ς'(Recanalization. . .)

(b) Record transformation for `Hospitalization` after the alignment with `Admissions`.

SocialSecurityNo	disease	week	month	year	ward	ICD-9-CM
BCDVHZ59S23F743S	Right vestibular deficit	2	1	2017	Emergency	386.10
PNPMZZ74H45H782P	Severe Appendicitis	8	2	2017	Emergency	540.9
PKTBMF36E14H842O	Recanalization of Crohn's Disease	15	3	2017	Emergency	555.0

(c) Resolution of the transcoding functions ς over the aligned `Hospitalization`

Figure 2.2 Data integration steps: intermediate schema alignment and transcoding transformation steps before providing the final result.

2.1.1 Structured Data integration: Integrating entities represented with different schemas

Given that both Data Warehouses require an associated schema towards which the data sources are transformed, and that most of the Data Warehouse literature relies on a multidimensional representation of relational databases (ROLAP), we can freely assume that Data Warehouses are structured data stores. We can now extend and adapt their use cases to the relational model, like the ones already presented in [GMP$^+$12]. The following example provides an use case that will also be restated for other data structures.

▸ **Example 1.** *A set of local health-care departments share some internal knowledge into a peer-to-peer Business Intelligence network. Within this network, such departments exchange medical records concerning the patients. Moreover, all such departments have different data representation for the same concepts.*

Suppose that one department wants to retrieve all the records about the cured patients in both its structure and in an external department: provide a uniform representation for the patients using the local representation format.

The local data warehouse provides a table `Admissions`*, as depicted in Figure 2.1a: we now want to grasp all the admissions from remote locations, which may store the same entities with different relation names and with different data schemas. An example is the* `Hospitalization` *relation in Figure 2.1b, where the ICD-9-CM*[2] *field refers to a machine-readable representation categorizing patients' diseases.*

After retrieving the two aforementioned tables to be integrated, the associated schemas[3] must be extracted and compared, as showed in Figure 2.2a: this preliminary step is required before actually integrating the data, because we must first detect which are the fields describing the same concepts in both relations. This comparison process at the schema level is called **alignment**, and it will be addressed in Section 2.3.2 on page 49.

We can subsume the schema alignment as follows: **(a)** each attribute having the same name will contain the same concept and, whether similar concepts are represented in different formats and representations, some transcoding function[4] ς is associated [GMP$^+$12]. Otherwise, a correspondence is required: **(b)** if terms between the two schemas are similar (e.g., synonyms), they are aligned as in **(a)**. Last, **(c)** if there is no perfect match, then either one attribute A is more general than the other B (A is a **supertype** for B [LN07], $A \sqsupseteq B$), or vice versa ($A \sqsubseteq B$); when $A \sqsupseteq B$, then the conversion is not always possible (e.g., if you have a `Week` number, you cannot map this information into a precise `Day` of the month), but the reverse process is always possible through a transcoding ς'. Please note that all the transcodings could be either inferred using other ontologies[5] or by using techniques exploiting artificial intelligence inference tools which suggest such relations. Therefore, the ways how such alignments may be provided are beyond the targets of the present thesis.

▸ **Example 1** (continuing from p. 20). *Regarding Figure 2.1, an ontology may detect that the* `Diagnosis` *allows to identify a* `Disease`*, and hence the two fields shall provide the same content. This scenario fits case a: since they represent a description, no translation is required; the* `ICD-9-CM`

[2] https://web.archive.org/web/20140212190115/http://www.who.int/classifications/icd/en/

[3] For both structured and semistructured data, the **schema** represents the "patterns" through which the data is represented using a specific data model. In this case, the tables' headers plus their associated data types represent their schema. See Chapter 3 for more complete details.

[4] It is represented by the `stigma` Greek letter: Ϛ, ϛ.

[5] See Section 2.3.1 for more details.

code offers a Diagnosis *for a disease and such code describes a disease: a transcoding from one concept to the other is then required.*

Last, since Day *is part of the* Week *and since each* Month *could have several* Weeks*, an ontology can infer the* Week *to which a* Day *belongs to from the information of* Day *and* Month*: this "alignment" step must be handled as described on case **(c)**.*

The final result for table Hospitalization *to match the schema of* Admissions *is then showed in Figure 2.2b: first, we associate to each field how such alignments have to be solved (Figure 2.2a on page 19), and then definitively solved (Figure 2.2c) before being merged through a "union" in the* Admissions *table (Figure 2.1c).*

2.1.2 Semistructured Data Integration: Integrating multiple relations into a common representation

In the previous subsection we saw one possible way of performing data integration, that is the integration of local data with external sources. Moreover, data was represented in a tabular form. Now, we want to integrate together *(i)* different concepts *(ii)* described using no explicit schema, *(iii)* while representing the final data with an uniform representation. The third condition also implies that no (global) matching schema is provided. In this scenario, we require that the schema must be extracted in a (semi)automatic fashion from the data sources. As we are going to address in Section 2.2.1, this is always possible because, to each datum stored in a field and using a specific data representation, we can always associate a **type**. Let us now change the scenario slightly:

▸ **Example 2.** *Suppose now to cope with complete medical records, which are represented in two distinct data source as semistructured XML documents with different associated schemas. The records shall track the patient from the admission ("Hospitalization") to the discharge from the hospital.*

Integrate such records belonging to both the remote hospital and local hospital, while preserving the whole information coming from the two distinct sources.

If we just focus on the semistructured representation provided from the data source in Figure 2.5b on page 25, we see that three distinct concepts are represented: the patients that entered the hospital (<patient>. . . </patient>*), whether this patient was cured (*<treatment>. . . </treatment>*) and which ward he attended (*<ward>. . . </ward>*). On the other hand, Figure 2.5a represents the patients (*<patient>. . . </patient>*) and a record associated with him/her (*<record>. . . </record>*). Given this data, we want to obtain the final result provided in Figure 2.6 on page 26, using an intermediate representation between the two.*

The data integration process seems quite hard at first because no schema is associated to the original data. The extraction of an associated schema is always possible [BLC+17] and, as a result, we're going to apply the schema alignment on top of such extracted representation. Figure 2.3 on the next page provides the associated schema to the two XML files in a JSON format[6]. At the schema level we observe that while schema (a) describes a person's full name through the Surname and Name tags, schema (b) only provides a name field. If we extend our schema comparison to the actual hospitalised patients' records in both medical structures, we could then infer that name in (b) contains both Name and

[6] Even though there are standard and conventional ways to associate schemas to XML files such as **XML Schema** [Vli02], **RelaxNG** and **DTD**, and given that it is always possible to convert an XML file into JSON, I here prefer to show the XML schema using a more compact JSON representation. Both XML and JSON are, anyhow, semistructured data representations.

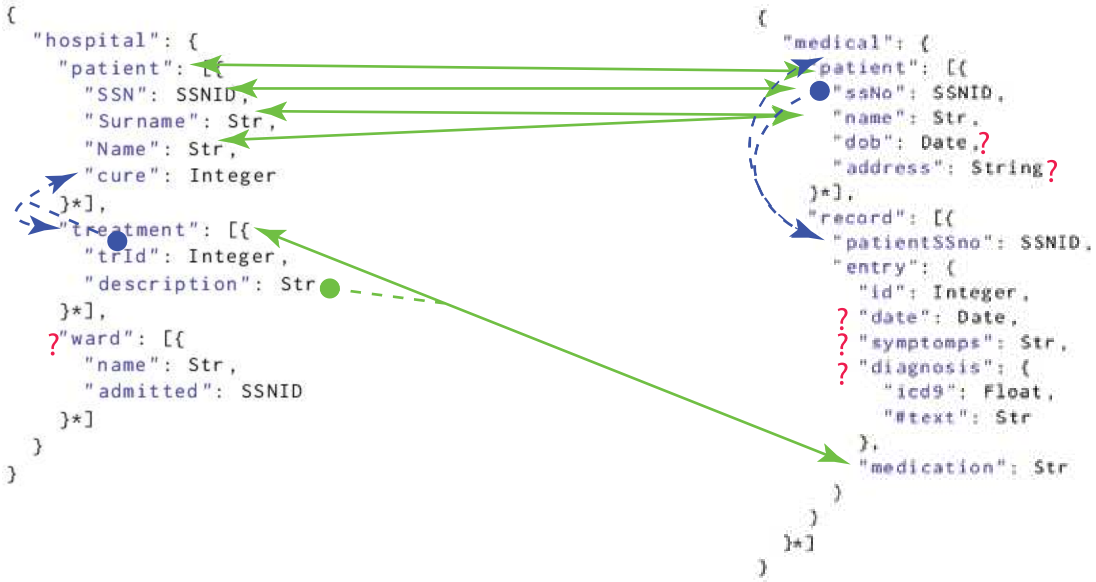
```
{
  "hospital": {
    "patient": [{
      "SSN": SSNID,
      "Surname": Str,
      "Name": Str,
      "cure": Integer
    }*],
    "treatment": [{
      "trId": Integer,
      "description": Str
    }*],
    "ward": [{
      "name": Str,
      "admitted": SSNID
    }*]
  }
}
```

(a) Representng the local clinical record using the schema in [Pog06].

```
{
  "medical": {
    "patient": [{
      "ssNo": SSNID,
      "name": Str,
      "dob": Date,
      "address": String
    }*],
    "record": [{
      "patientSSno": SSNID,
      "entry": {
        "id": Integer,
        "date": Date,
        "symptomps": Str,
        "diagnosis": {
          "icd9": Float,
          "#text": Str
        },
        "medication": Str
      }
    }*]
  }
}
```

(b) Representing the local clinical record using the schema in [MFK01].

Figure 2.3 Representing the schema associated to the XML files using the notation provided in [BLC+17] for JSON documents. Instead of associating values to each property, we associate properties to their types. Arrays of `T` elements are represented with the `[T*]` syntax, where `T` may represent another record, representing a composite type.



```
{
  "hospital": {
    "patient": [{
      "SSN": SSNID,
      "Surname": Str,
      "Name": Str,
      "cure": Integer
    }*],
    "treatment": [{
      "trId": Integer,
      "description": Str
    }*],
  ?"ward": [{
      "name": Str,
      "admitted": SSNID
    }*]
  }
}
```

```
{
  "medical": {
    "patient": [{
      "ssNo": SSNID,
      "name": Str,
      "dob": Date,?
      "address": String?
    }*],
    "record": [{
      "patientSSno": SSNID,
      "entry": {
        "id": Integer,
     ? "date": Date,
     ? "symptomps": Str,
     ? "diagnosis": {
          "icd9": Float,
          "#text": Str
        },
        "medication": Str
      }
    }
  }*]
}
```

(a) Complete alignment process of the two schemas associated to the XML files. The green lines remark the alignments between the schemas, the blue lines mark the key detection within a local schema. When these terminological correspondences (edges) do not directly connect identical values which may appear in internal fields, such field is marked with a filled circle. The red question marks underline the fields that are missing in the other schema, and that will be filled with missing fields.

```
{
  "medical␣hospital": [{
    "patient": [{
      "SSN": SSNID,
      "Surname": Str,
      "name": Str,
      "dob": Date,
      "address": String,
      "cure_treatmentId", Integer
    }*],
    "treatment_entry": [{
      "id": Integer,
      "patientSSno": SSNID,
      "date": Date,
      "symptomps": Str,
      "diagnosis": {
        "icd9": Float,
        "#text": Str
      },
      "medication": Str
    }*],
    "ward": [{
        "name": Str,
        "admittedPatient": SSNID
    }]
}
```

(b) Merged schema resulting from the hschema alignment process.

Figure 2.4 Alignment process between the XML schema, and final XML schema integration

Surname in (a). Where this is not possible, we could always assume that we have a personal data bank of *names* and *surnames*, through which detect which part of name contains a name and which contains a surname. Schema (b) also provides some more fields that are not in (a), such as the date of birth (dob) and the patient's address. By using an external ontology, we may also know that medication in (b) corresponds to treatment in (a) and, given that the only field which has an associated Str(ing) in it is description, then medication and treatment shall both contain the same type of content. We can also infer that record is a supertype of treatment because the first contains more fields than the latter one.

As a second step, we can make some assumptions on which fields represent a key field within each single data source, by comparing the data in Figure 2.5b on the facing page and the associated schema in Figure 2.3a. We notice that cure is a synonym for treatment, and hence they can refer to the same concepts: by comparing the data sources, values in cure correspond to values in trId which has a functional dependency with treatment. We can infer that admitted refers to a SSN because they have the same type; if more sophisticated tools are available (e.g., ontologies) by knowing that an hospital can admit a patient, we can also infer that SSN is a key for patient, which contains the SSN field, and that admitted refers to a patient. This assumption could also be confirmed by looking at data values.

Figure 2.4a provides all the matches inferred for the schema alignment process, including the previous steps described in words. This alignment also allows to extend the "syntactic" semistructured schema integration approach presented in [BLC+17], and it consequently provides the merged schema depicted in Figure 2.4b. At this point, we can transform both sources to match the provided schema using transcoding functions, and then provide a global representation of such integrated data as already presented in Figure 2.6 on page 26. As the last step, we could integrate all the fields that pertain to the same entity using clustering techniques (in this case, we can merge all the patients having the same SSN id). Section 2.2 addresses this last part while focusing on dimensionality reduction and data cleaning processing. Moreover, if the use ontologies which are powerful enough to express that *"an Italian SSN contains information concerning the date of birth of a person"*, then such information could be easily imputed using some associated transcoding functions.

By comparing the alignments process for structured data and the one for semistructured data, we observe that the schema alignment process is data representation independent, and that all the computational steps reduce to a comparison of either fields (in the relational case) or compound fields (in the semistructured one) defining *entities*, which appear in two different schemas. After the former comparisons, correspondences are found. Section 5.4.2 on page 143 is going to further investigate this topic, where we're going to answer the following question: *"is it possible to express both schemas and alignments within the same representation?"*.

2.1.3 Structured and Semistructured data integration: schema alignment as a data cleaning step

Given the considerations of the former paragraph, we would like to use the outcomes of the data integration process presented in Section 2.1.1 as imputations for the missing values coming from the semistructured integration phase. In particular, we want that such data can be integrated with the ward and the ICD-9-CM pieces of information coming from the Hospitalization table. We must now use both the source data (Figure 2.1a on page 19) and the transcoding functions generated from the alignment between the two schemas

```
<medical>
  <patient ssNo="BCDVHZ59S23F743S">
    <name>Bocedi, Venhz</name>
    <dob>23/11/1959</dob>
    <address>via San Biagio 1, Morozzo, Cuneo</address>
  </patient>
  <record>
    <patientSSno>BCDVHZ59S23F743S</patientSSno>
    <entry id="1">
      <date>01/01/2017</date>
      <symptomps>fluid draining from the ear,
       trouble swallowing</symptomps>
      <diagnosis icd9="210.2">Right parotid
       neoplastic formation</diagnosis>
      <medication>Radiotherapy</medication>
    </entry>
  </record>
</medical>
```

(a) Representing the local clinical record using the schema in [MFK01].

```
<hospital>
  <patient>
    <SSN>BCDVHZ59S23F743S</SSN>
    <Surname>Bocedi</Surname>
    <Name>Venhz</Name>
    <cure>505</cure>
  </patient>
  <treatment>
    <trId>505</trId>
    <description>Epley maneuver</description>
  </treatment>
  <ward name="Emergency">
    <admitted>BCDVHZ59S23F743S</admitted>
  </ward>
</hospital>
```

(b) Representing the external clinical record using the schema in [Pog06].

Figure 2.5 Representing two possible medical records for the same patient coming from different hospitals.

```
<medical_hospital>
  <patient>
    <SSN>BCDVHZ59S23F743S</SSN>
    <Surname>Bocedi</Surname>
    <Name>Venhz</Name>
    <dob>23/11/1959</dob> <!-- from SSN-->
    <address />
    <cure_treatmentId>505</cure_treatmentId>
  </patient>
  <patient>
    <SSN>BCDVHZ59S23F743S</SSN>
    <Surname>Bocedi</Surname>
    <Name>Venhz</Name>
    <dob>23/11/1959</dob>
    <address>via San Biagio 1, Morozzo, Cuneo</address>
    <cure_treatmentId>1</cure_treatmentId>
  </patient>
  <treatment_entry>
    <id>505</id>
    <patientSSno>BCDVHZ59S23F743S</patientSSno>
    <date />
    <symptomps />
    <diagnosis />
    <medication>Epley maneuver</medication>
  </treatment_entry>
  <treatment_entry>
    <id>1</id>
    <patientSSno>BCDVHZ59S23F743S</patientSSno>
    <date>01/01/2017</date>
    <symptomps>fluid draining from the ear,
    trouble swallowing</symptomps>
    <diagnosis icd9=”210.2”>Right parotid
      neoplastic formation</diagnosis>
    <medication>Radiotherapy</medication>
  </treatment_entry>
  <ward name=”Emergency”>
    <admittedPatient>BCDVHZ59S23F743S</admittedPatient>
  </ward>
</medical_hospital>
```

Figure 2.6 Expected outcome of the XML data integration. Missing values are represented by tags with no contents.

```
<medical_hospital>
  <patient>
    <SSN>BCDVHZ59S23F743S</SSN>
    <Surname>Bocedi</Surname>
    <Name>Venhz</Name>
    <dob>23/11/1959</dob>
    <address />
    <cure_treatmentId>505</cure_treatmentId>
  </patient>
  <patient>
    <SSN>BCDVHZ59S23F743S</SSN>
    <Surname>Bocedi</Surname>
    <Name>Venhz</Name>
    <dob>23/11/1959</dob>
    <address>via San Biagio 1, Morozzo, Cuneo</address>
    <cure_treatmentId>1</cure_treatmentId>
  </patient>
  <treatment_entry>
    <id>505</id>
    <patientSSno>BCDVHZ59S23F743S</patientSSno>
    <date>NULL/01/2017</date>
    <symptomps />
    <diagnosis icd9=”386.10”>Right vestibular deficit</diagnosis>
    <medication>Epley maneuver</medication>
  </treatment_entry>
  <treatment_entry>
    <id>1</id>
    <patientSSno>BCDVHZ59S23F743S</patientSSno>
    <date>01/01/2017</date>
    <symptomps>fluid draining from the ear,
    trouble swallowing</symptomps>
    <diagnosis icd9=”210.2”>Right parotid
      neoplastic formation</diagnosis>
    <medication>Radiotherapy</medication>
  </treatment_entry>
  <ward name=”Emergency”>
    <admittedPatient>BCDVHZ59S23F743S</admittedPatient>
  </ward>
</medical_hospital>
```

■ **Figure 2.7** Merging the two relational tables from Subsection 2.1.1 with the XML document resulting from Subsection 2.1.2. This data integration step fills in the missing values marked with ?. After analysing the schema alignments paired with the to-be-aligned data, we can draw more guesses than the ones initially formulated for providing Figure 2.6 on the preceding page.

(Figure 2.2a on page 19). Firstly, we still have to create correspondences between each XML `treatment_entry` and a record belonging to either `Admissions` or `Hospitalization`. As previously noticed, it is not possible to retrieve the precise day of the week of the `Admission` of a patient if no further precise information is known: for this reason, we're going to use the following transcoding function, where the day information is coded through a day null value, 00:

$$\varsigma^{-1}(week, month, year) = \text{“}00/month/year\text{”}$$

Concerning the data coming from the external sources, we can still associate a `ICD-9-CM` code to each `disease` representing a `diagnosis` through the ς' given above.

Figure 2.7 provides the outcome of this alignment and integration phase: as we can see, the data imputation phase required for data cleaning may resemble the data integration steps (see Section 2.2). The only missing pieces of information are the patient's address (that could be easily retrieved after another data cleaning process after which the `patient` record will be merged), and the symptoms related to the "Right vestibular deficit".

If no symptoms are provided, we can now ask ourselves if it still possible to retrieve this information from full-text medical books: which is the best way to represent such information in order to answer the question "*Which are the symptoms for balance disorders?*"?

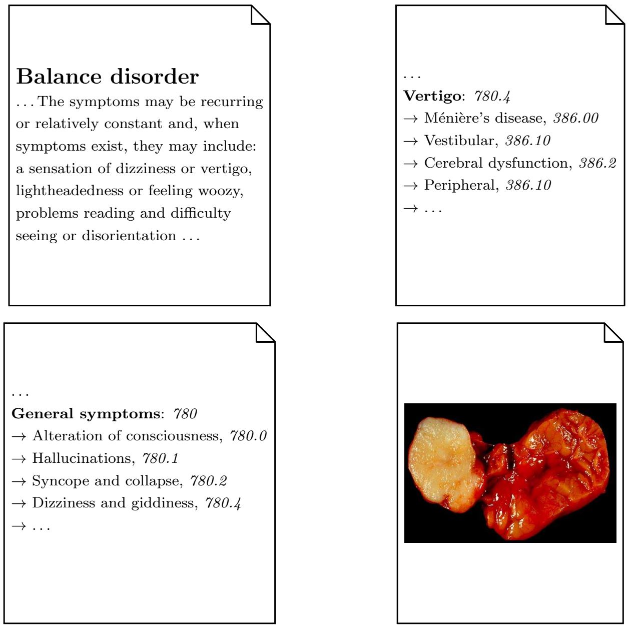
2.1.4 Integrating unstructured data via semistructured representation

Unstructured data represents digital information that cannot be immediately used as pieces of information until its transformation in a semantic dependent form is provided [RH08]. Unstructured data usually has no associated description for interpreting such information. Figure 2.8 provides some examples of unstructured contents: the index at bottom left represents an association between some medical terms represented in a hierarchical form and some ICD-9-CM codes, while the picture represents a tumour of the salivary glans. As we will see in Section 3.3 on page 70, full-text content shall be represented either in a purely syntactical structured form (*spans*), or in a semistructured representation. Such representations include graphs (*universal dependencies graph*) and nested graphs[7] (proposed by this thesis in Section 6.3.4 on page 183). Suppose now that we want to integrate the three textual representations altogether: we cannot mix them at the textual level because full-text documents could be represented in different formats (txt, doc, RDF, wiki markup or pdf) and may also express different concepts requiring different interpretations. As a consequence, the best way to integrate unstructured data is to first provide them some structure reflecting their semantics, and then try to merge the data.

▸ **Example 3.** *We now want to merge the two indices presented in Figure 2.8 to create an ontology allowing the detection of clusters of similar diseases*[8]*. We cannot directly combine the two indices into one single file, because each file must be processed differently; while the taxonomy is a tree where all the terms point to the most general term, the alphabetical index is a graph where each general term shall point to the most specific sub-item. In the last scenario, each of these terms are disambiguation terms and directly depend on the main synset. In the previous indexing structure, distinct terms could be related to the same concept represented by a unique ICD-9-CM code: e.g., both "vestibular vertigo" and "peripheral vertigo" lead to the same concept, labelled as 386.10, classified within the taxonomy as "Peripheral vertigo, unspecified" (not showed).*

[7] Since both nested graph data integration is proposed for the first time by this thesis, we refer to Section 2.1.5 on the next page for further details.

[8] `https://rebrand.ly/githu52932`



Balance disorder

. . . The symptoms may be recurring or relatively constant and, when symptoms exist, they may include: a sensation of dizziness or vertigo, lightheadedness or feeling woozy, problems reading and difficulty seeing or disorientation . . .

. . .
Vertigo: *780.4*
→ Ménière's disease, *386.00*
→ Vestibular, *386.10*
→ Cerebral dysfunction, *386.2*
→ Peripheral, *386.10*
→ . . .

. . .
General symptoms: *780*
→ Alteration of consciousness, *780.0*
→ Hallucinations, *780.1*
→ Syncope and collapse, *780.2*
→ Dizziness and giddiness, *780.4*
→ . . .

Figure 2.8 Different possible types of unstructured information: full-text sentences from Wikipedia (top left), alphabetical indices (top right), taxonomy indices (bottom left) and pictures (bottom right).

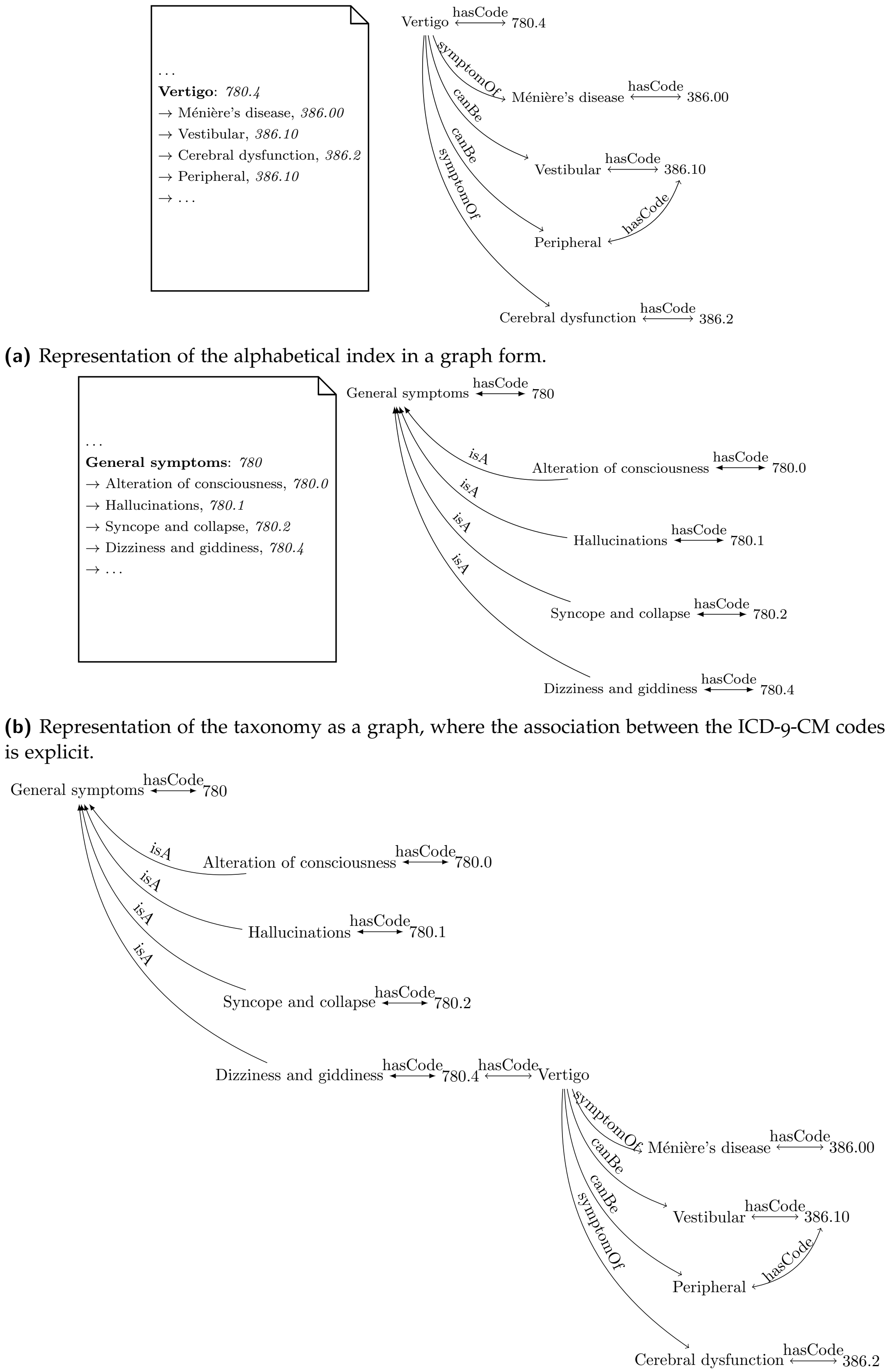
As a consequence, different graph representation techniques shall be adopted for distinct files: Figure 2.9 represents an example of how such indices could be first represented (a and b), and then merged into one single graph (c), thus providing an ontology. The integration of the two graphs is merely performed via a union of the vertex and edge set.

The graph representations provided in the last example are not the sole one possible for full-text corpora: Figure 2.10 provides an example of the NLP interpretation of the top left full-text from Figure 2.8, that is also machine-readable. Section 3.3 on page 70 is going to present more insights concerning the need for an intermediate representation of unstructured data.

2.1.5 Aligning (Nested) Graphs

If we shift the schema alignment process from (semi)structured data to graphs, it is also possible to extend the schema alignment techniques to graph schema alignments. Given that graphs are a specific case of semistructured data (because they have no strict associated schema), it is also possible to represent full-text as both data, schema and queries. Graph schema alignments are the result of the query answering process: in this scenario, we align the actual data vertices and edges instead of the data schema, and where vertices may represent variables to be instantiated for solving the desired query.

▸ **Example 4.** *Let us leave the healthcare data integration scenario. Suppose that we want to answer the following question: "In May 1898 Portugal celebrate the 400th anniversary of the arrival of **this explorer** in India" [oRD12], where **this explorer** is the subject that has to be found, and hence the "variable" to be instantiated. Since this is a very general question, we have to use some full-text corpora, like the one from Wikipedia. From this full-text sources, we can extract different sentences from different pages. At this step, we want to check which part of the text matches with*



...
Vertigo: *780.4*
→ Ménière's disease, *386.00*
→ Vestibular, *386.10*
→ Cerebral dysfunction, *386.2*
→ Peripheral, *386.10*
→ ...

(a) Representation of the alphabetical index in a graph form.

...
General symptoms: *780*
→ Alteration of consciousness, *780.0*
→ Hallucinations, *780.1*
→ Syncope and collapse, *780.2*
→ Dizziness and giddiness, *780.4*
→ ...

(b) Representation of the taxonomy as a graph, where the association between the ICD-9-CM codes is explicit.

(c) Integration of the two indices as a common graph, thus representing an ontology.

Figure 2.9 Representing full-text contents as graphs.

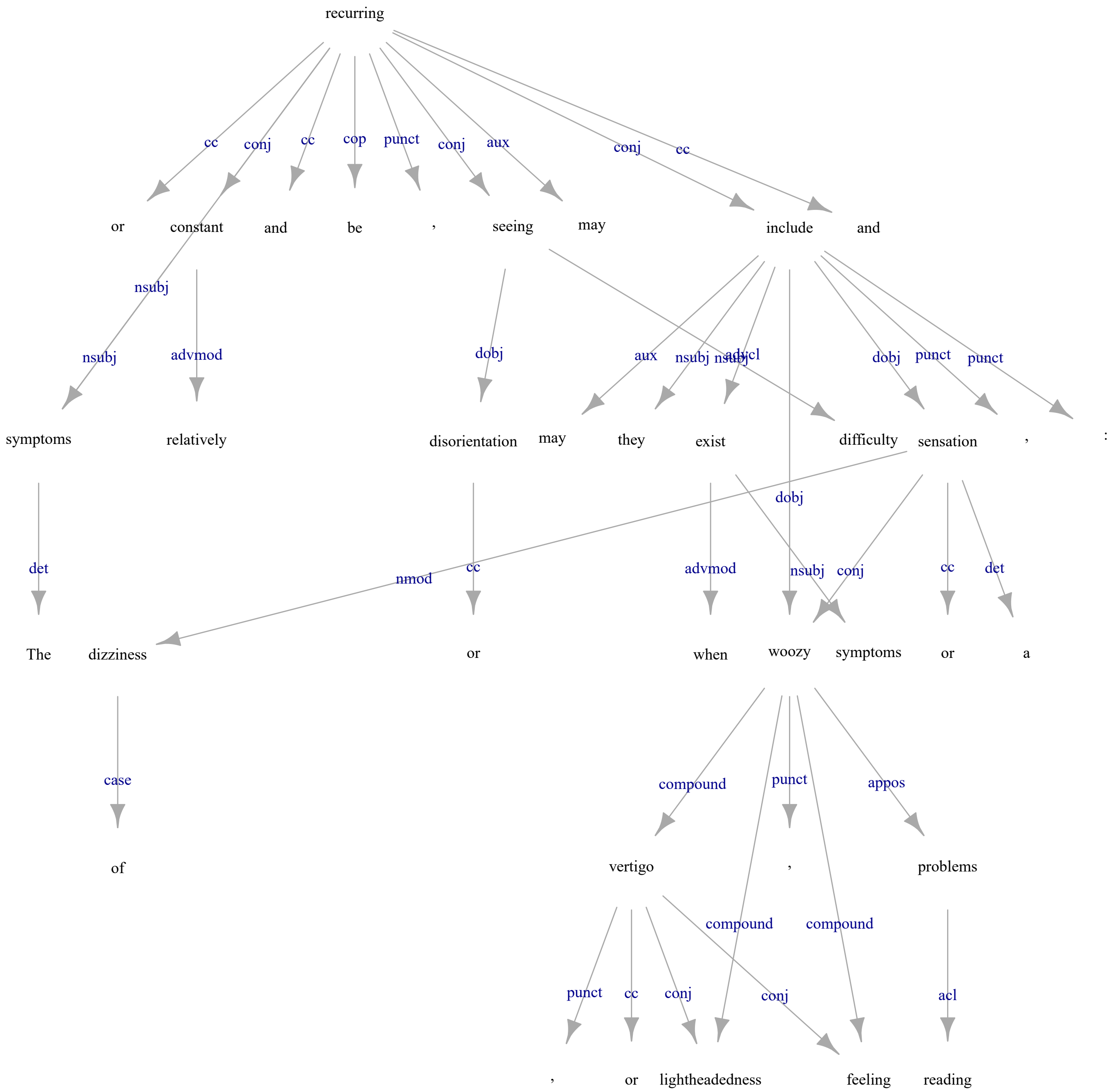


Figure 2.10 Dependency graph of the wikipedia fulltext in Figure 2.8 obtained via the Stanford NLP Library (See my source code at `https://rebrand.ly/unibo7252c`). Each vertex represents one word within the full-text document, while each edge expresses a part of speech grammatical function, called *universal dependency* (`http://universaldependencies.org`) [dMDS$^+$14], which may also be (human) language dependant (i.e., different languages may provide a different set of language dependencies).

our question. To do this we have to first provide a nested graph representation of both the question (Figure 2.11b on the facing page) and for the two candidate answers (a and c). As we can see, nested graphs allow to nest whole concepts, such as "the arrival of this explorer in India" as one single vertex, thus allowing to draw an edge between the "400th anniversary" and this other object.

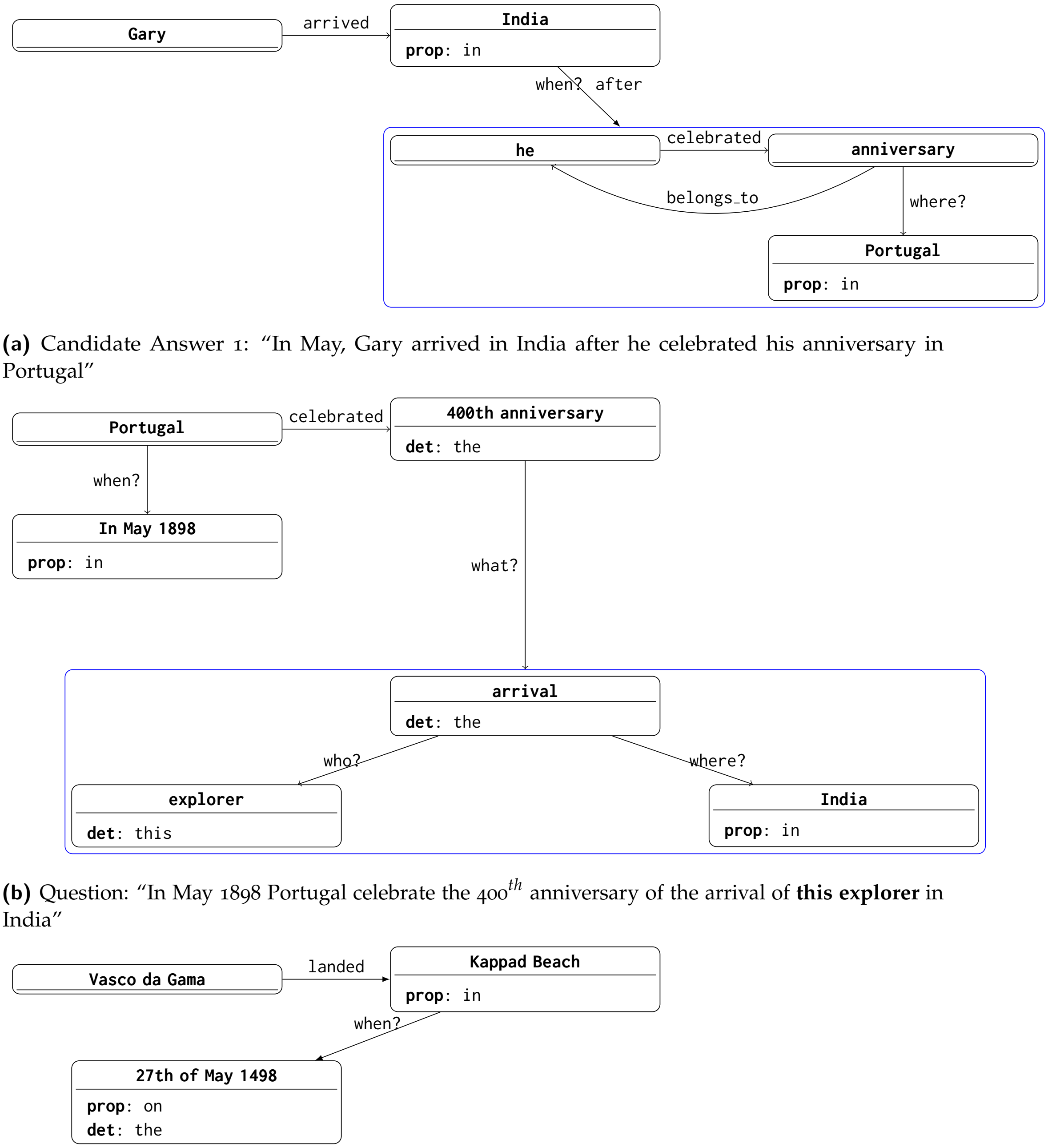
Even in this case, each graphs' node has neither an associated schema – that changes from question to question – nor a type associated to each vertex and edge. Nevertheless, we can first provide some types to the question: we have that `Portugal` *is a* **Place** *as well as* `India`*, and* `explorer` *is elected as a type itself since it is not a simple term of the graph. The last one also represents the goal of our question. All the verbs and nouns expressing motions are marked as* **Action** *(e.g.,* `celebrated` *and* `arrival`*), while more generic terms are associated to* **Concept***, such as Anniversary. Temporal pieces of information can be marked with a* **Time** *type. Before starting the alignment process, we must ask ourselves which elements are considered as* `explorers` *or at least* **Person***s in the two candidate answers. After using an ontology such as* ***DBpedia****, we know that* `Vasco da Gama` *is for sure an explorer but, using the open world assumption, we cannot say that* `Gary` *is not for sure an explorer, and so we associate to him a* **Person** *type. In the next phase, we want to associate the place* **Place** `India` *to all the other geographical pieces of information within the candidate answers: while* `India` *perfectly matches as both type and content in the first candidate answer, we find* `Kappad Beach` *in the second answer. Even in this case, after using DBpedia we can infer that* `Kappad Beach` *is a* ***part-of*** `India`*, and hence in both cases we have a perfect match. At this stage, we can also try to find the matches between the temporal vertices: while the first candidate answer has no temporal information, the first one has an information that, on the other hand, is very hard to match with the one contained in the question. At this step, we could think to provide the original question some refinements: if we have the availability of an ontology matching the concept of* `400th anniversary` *to a function:*

$$anniversary(number, when) = when - number$$

then we can infer that if at time when we celebrated the number anniversary, then the concept shall refer to a fact occurred on the year "when – number". In this case, we will obtain $1898 - 400 = 1498$*, and so we can extend the nested vertex within the question with a* `when` *edge and the resolved time,* `May 1498`*: finally, we have mined a further correspondence between the temporal aspect.*

After finding all the correspondences between the vertices and nested vertices, we have to map the **Action** *entities either to edges (since edges in this graph could also represent verbs) or other entities: in this case we have that the type information is not enough because it is too general, and hence we require some term similarity, that can be even in this case achieved using term ontologies such as* ***Babelnet****. At this stage, we can make correspondences between the terms* `landed`*,* `arrival` *and* `arrived`*, and hence the* `arrival` *matching with* `arrived` *could be transformed into a verb, and hence into an edge. Moreover, the result of the alignment process including such graph transformations is presented in Figure 2.12. We can now finally perform some approximated graph matching techniques [VMT15, AGG+15], after which we can observe that the second candidate answer contans all the information we are looking for and, since* `this explorer` *and* `Vasco da Gama` *match and have the same type, we can say that* `Vasgo da Gama` *is the answer we were looking for from the very beginning.*

This last example showed that alignment techniques could be used not only for aligning schemas, but also for finding correspondences within ambiguous data representations. As a consequence, this example shows that graphs could be used to represent both schemas and data, and hence could be used to represent two different and distinct concepts. Consequently, we need a language expressing graph schemas, thus generalizing the schema language for semistructured documents in [BLC+17]. Moreover, it has been also showed



(a) Candidate Answer 1: "In May, Gary arrived in India after he celebrated his anniversary in Portugal"

(b) Question: "In May 1898 Portugal celebrate the 400th anniversary of the arrival of **this explorer** in India"

(c) Candidate Answer 2: "On the 27th of May 1498, Vasco da Gama landed in Kappad Beach."

Figure 2.11 Representing the question (b) and two possible candidate answers (a and c) using nested graphs. Candidate answer 2 is correct.

that graphs could also represent query languages for graphs [CM90a, CM90b, FLM+12, GPG14]: as we will se in the next section, Data, Model and (Query) Language represent three distinct levels within usual modelling languages, while graphs allow to collapse into one single representation.

Last, pictures can be also represented as (semi)structured data: object categorization techniques [GB10] such as region-based segmentations [GGK09] can be applied to extract interesting features from the pictures. By doing so each picture is automatically associated to the features that it contains. Within the e-Health scenario, such techniques have been applied to tumour detection and classification [RJKK15, RJ16], where the benign forms are separated from the malignant ones. Each fragment allowing such categorization could be then stored alongside with the classification's outcome, thus providing a way to transform unstructured data to structured pictures providing informations. Moreover, pictures could be represented as hypergraphs [BG05] and hence also as nested graphs.

We can finally observe that, prior to allowing the integration of graphs with structured and unstructured data, we need to find a common representation for all the provided datasets. Hence, we have to discuss which is the best and general data representation allowing to represent both nested component and edges, and why such features are important and relevant. For this reason, I refer to Chapter 3 on page 55 and 5 on page 119. Moreover, a more general approach considering data at different nesting levels is going to be discussed in Section 5.4 on page 139 after describing our proposed nested graph data model.

2.2 In-Database Integration

As observed in Section 2.1.3, the data integration step can also be used to perform data cleaning and duplicate removal. This also implies that, when a uniform data representation for all the data sources is provided (either structured or semistructured), such data cleaning steps can be also performed within the same data model and query language [LN07]. It is remarked that SQL lacks a proper support for data cleaning it its (Commercial) Off-the-Shelf dialects: such operations must be necessarily performed at the software level, thus failing on optimizing such tasks within the same environment.

We now outline two limitations of such language: first, SQL does not allow unions between tables having different schemas (`Outer Union`). This is the first required pre-processing step before performing some further cleaning operations, as described in the incoming paragraph. This union requirement could be easily met if we change the data model and chose a semistructured representation, which is schema flexible. We start to discuss this general data model from Section 2.2.1.

Second, SQL lacks functions clustering similar entities together (or which are able to accept collections of data collections as an input) and associating to each cluster (or data collection) one single object and – at the same time – preserving the original content. This is due both to data representation problems and query languages' limitations: for example, the `Group By` clause in SQL can only subsume records by aggregating them only if they share the same values for a certain set of chosen fields. For this reason, we will introduce a more general aggregation operation in Section 2.2.2 on page 40.

2.2.1 Preliminaries: towards a uniform data representation

In this section we want to show that the MetaObject Facility (MOF) data model (used for meta-modelling) can represent different data abstraction levels.

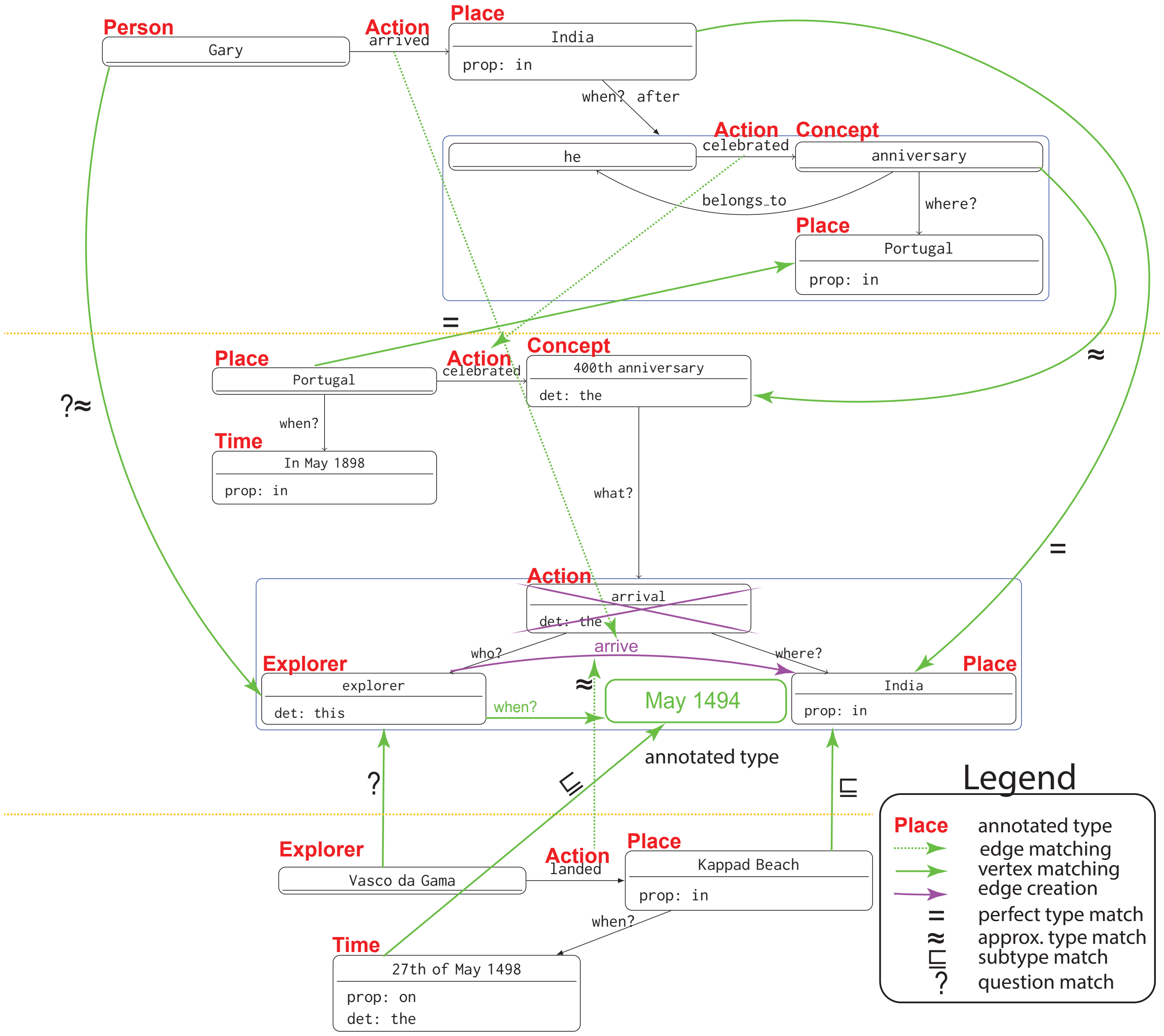


Figure 2.12 Aligning the query to the candidate answers through nested graphs.

The origins of meta-modelling could be retrieved in [omg96]: the OMG group aimed to develop a type system for a distributed computation system (CORBA), in order to manipulate data via "a common, integrated set of services". Through the definition of the **MetaObject Facility**, they wanted to distinguish between data, data collections (*types*) and properties hold by such data collections. Even though it was designed for Software Engineering, such hierarchy could be also applied to database systems [ACB06, LZ09] and the Semantic Web [BAdCG16].

▸ **Definition 1** (MetaObject Facility). *The **MetaObject Facility** relies on different layers representing different abstraction (and hence, different **abstraction levels**). The first layer, simply called* Data *($D = D_O \cup D_R$), contains both objects and relationships. In particular, an **object** $o \in D_O$ is an aggregation of **attributes** and **values** that provides an information. An object composed of different objects is referred as **composite**. **Relationships** $r \in D_R$ are n-ary association between objects. Relationships could also have a direction (directed relationships) or not (in this case they are called "undirected").*

The second layer is called Model *(M) and it is a simplified representation of reality. It contains the **types**, that are sets of objects $\mathcal{T} = \{ o_1, \dots, o_n \}$ satisfying a property p, written[9] $T(p)$, that is also called **schema** in the data science community[10]. Hereby, one object could belong to more than just one type (e.g.,[11] $o \in T_1 \wedge o \in T_2$), and hence satisfy more than just one property.*

The third layer is called MetaModel *(MM), and it contains the definition of the types and of the relationships used at lower levels: in particular it contains **meta-object**s describing properties p for given types $\mathcal{T}$.* ◂

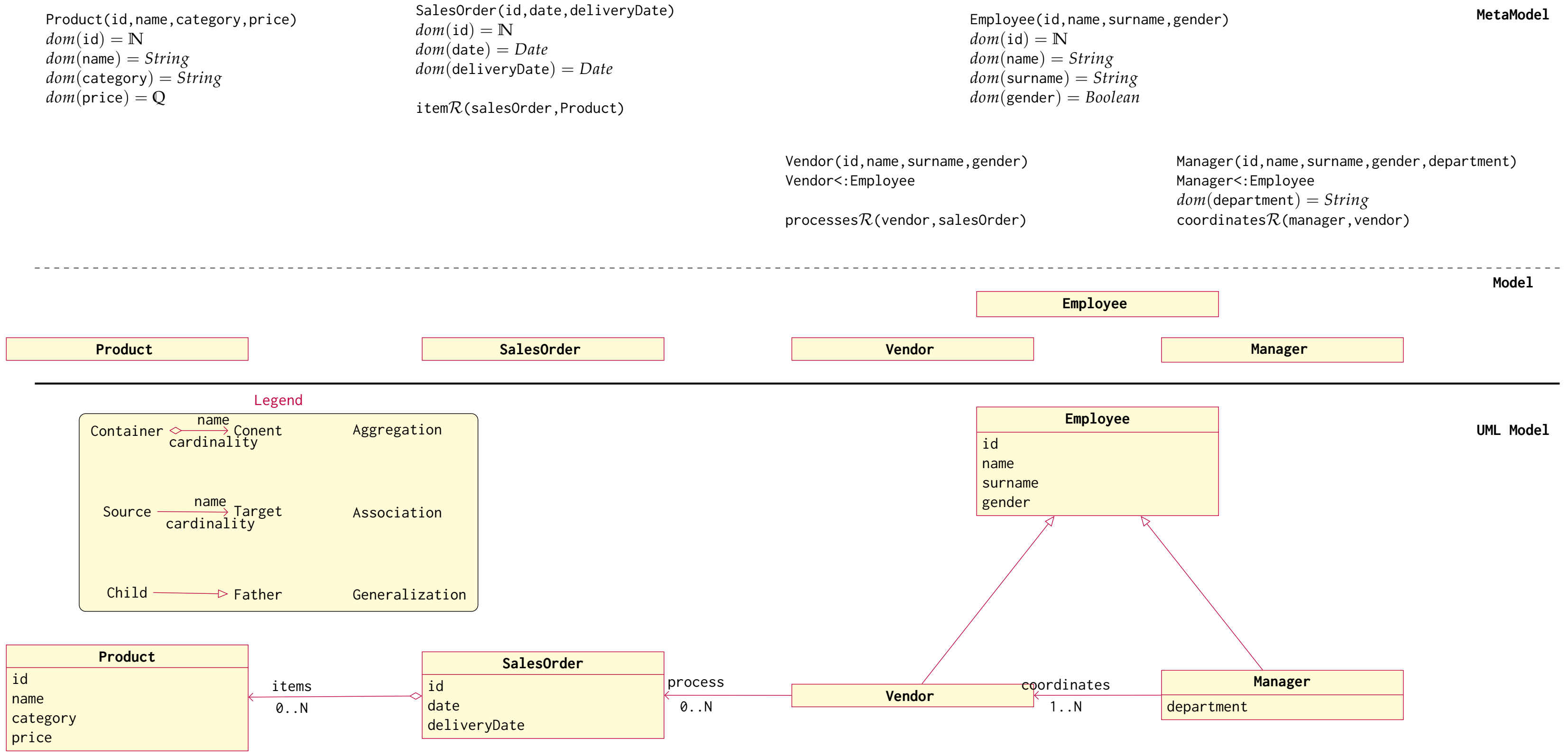
Besides of the very general representation of both entities and relationships, such model allows to point out at which abstraction level a query language must be located, thus introducing the next topic on ontologies. Moreover, its general definition of "value" allows it to be either an atomic value, or a collection of values, and each value could be also an object.

Despite the aforementioned definition, there is no universal acclaim of what a MetaModel should be. [HS12] points out some alternative approaches that have been followed on this regard (e.g., for the UML modelling language [OMG11b] extending the MOF): either **(i)** the MetaModel is the modelling language itself, allowing to outline the Model and its instance, or **(ii)** The MetaModel is a model for the modelling language. The choice of UML of class diagrams that could be used on both the Model and the MetaModel level causes a limit in the expressive power of the meta-modelling language, that could not be used to define more advanced properties (Figure 2.13) and, for example, cannot lead to the definition of a query language. Hereby, we will continue to use the MetaModel definition that was previously provided: as a consequence, Model and MetaModel do not abstract from the data structures, but characterize the properties hold by the same initial data.

[9] Originally, [omg96] used the inverse notation p_T, thus associating to each property a given type, and not vice versa. On the other hand, we use the notation $T(p)$ which resembles the notation in relational databases, where T is the relation and the predicate p is the associated schema. Moreover, in type theory [Pie02] such relation is expressed as $T :: p$, because there is a correspondence between the data types to the types in type theory, and between such (data) predicates to the sorts expressing the well-formed types.

[10] At this level relationships cannot be defined among types: this is one of the differences between Model as in the MetaObject Facility and Ontologies (Section 2.3.1) or UML Model (Figure 2.13).

[11] In this case I choose to adopt the set theory notation, widely used in relational databases to express that a tuple belongs to a relation ($t \in r$). On the other hand, in type theory an object is called a term, and to express that a term o belongs to a type T the notation $o\colon T$ is used [Pie02].



■ **Figure 2.13** Comparing the UML model representation and the one provided in Definition 1. The white diamond represents **part-of** relations and the white arrow represent **is-a** relations. As we could see, the UML Model already provides a structured notion of the types, since the type definition and the schema is not separated as in the MetaObject Facility. Hereby, we will continue to use the MetaObject Facility separation between Model and MetaModel, thus allowing to provide a data representation which is not schema related. In particular, we express schema (data types' properties) as in the relational model, and use the Datalog syntax for expressing relationships among types.

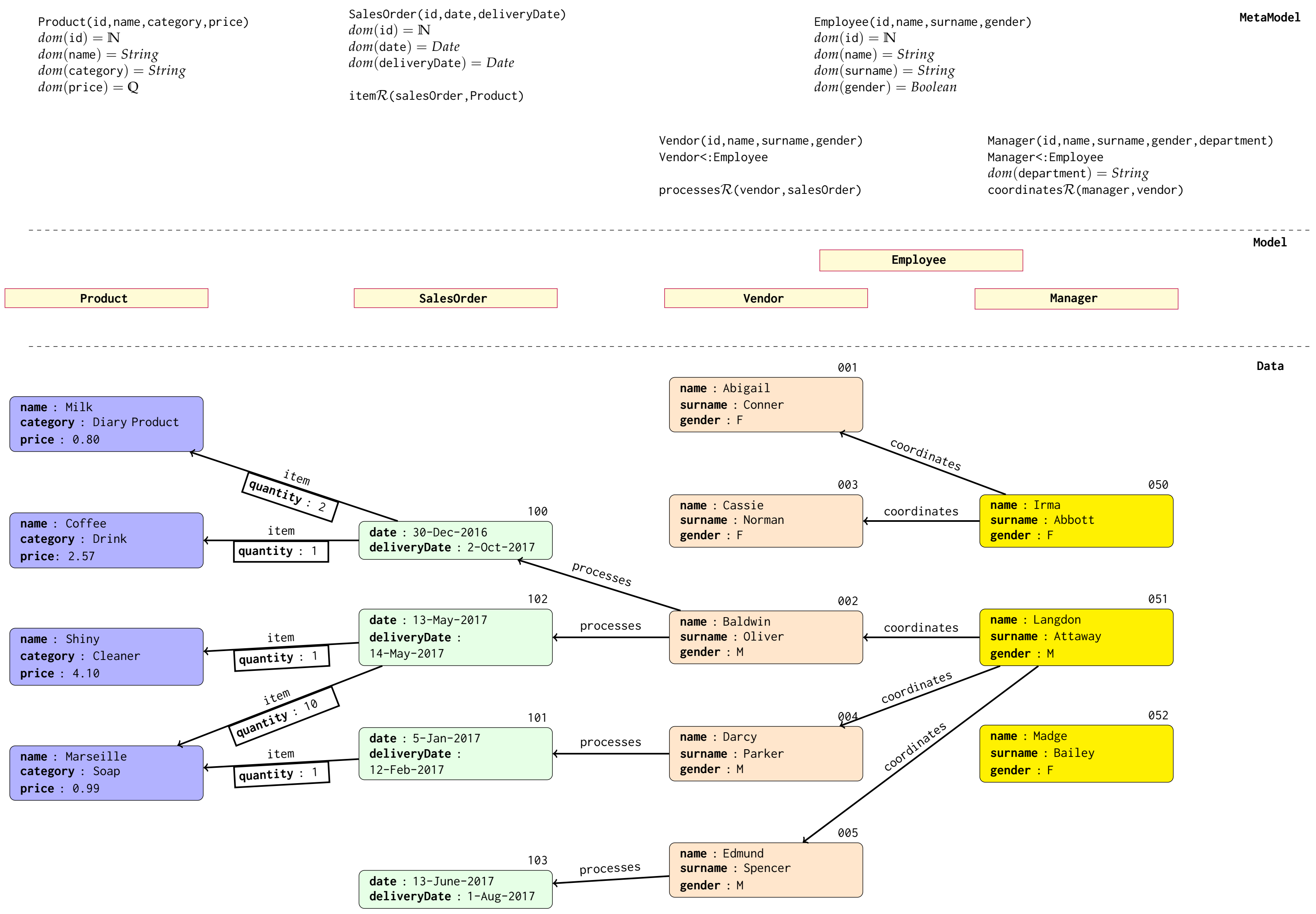


Figure 2.14 Complete MetaObject Facility example. Each object in the Data layer which has a given type (provided on the top label in bold) is associated to the class in the Model layer with an α relation.

▸ **Example 5.** *Figure 2.14 provides an example of all the MetaObject Facility layers for a vending company use case scenario. In particular, we represent two types of* `Employees`*,* `Vendors` *and* `Managers`*. The latter ones coordinate the former ones, while the first ones can process* `SalesOrders` *order containing* `Products`*. One* `Manager` *could* `coordinate` *one or more* `Vendors`*, which could* `process` *some* `SalesOrders` *composed of several* `items`*, which are* `Products`*. All the objects sharing the same type are arranged in the same column where their corresponding type at the* Model *level is located, on top of which its properties at the* MetaModel *are provided.*

As we could se from the former example, such model allows to represent a graph: each object could be considered as a vertex, while each relationship as an edge. As a result, we have that even edges must be represented as primary concepts within the MetaObject Facility. Hereby, even each relationship should be a primary concept within this modelling definition, and hence each edge should have an associated type, as it happens in current graph databases (compare with edge *labels*). On the other hand, this already happens within the UML modelling language [OMG11b, Obj11].

Given that the schema and all the properties pertaining to the types are expressed as properties, it is now easy to define any query Language $\mathcal{L}_{MM}$ on top of the MetaModel MM through which express the schema of the types at the Model level by abstracting from the data representation. The fact that $\mathcal{L}_{MM}$ includes MM is confirmed by the fact that *queries* could be represented as mere combinations of to-be-satisfied properties, as well as schema representations [Len02]. Please note that such query Language ($\mathcal{L}_{MM}$) does not belong to the original MetaObject Facility model, and that it has been introduced at this level for the first time within this thesis for modeling purposes.

Between such abstraction layers only an “instance-of” function[12] α is allowed to connect each lower layer to the immediately upper one [HS12]; such kind of relation is not allowed between instances of the same layer, and hence it could not be used to provide aggregations within the Data layer. As a consequence, α should be denoted by the following expression:

$$\alpha\colon D \uplus M \to M \uplus MM$$

where $\uplus$ provides the disjoint union between two sets, thus allowing to map each element of D into one of M, and similarly for M and MM. Such function is defined in literature as follows:

▸ **Definition 2.** *An* ***abstraction function*** *is an α relation that is defined as follows:*

$$MM = \alpha(M) = \alpha(\alpha(D)) \tag{2.1}$$

Such α could be seen as the following transformation between objects at different layers of abstraction [Kö6]:

$$\tau \circ \alpha' \circ \pi \tag{2.2}$$

where π is a projection function that reduces the informative content of the given object, α' provides a further abstraction (i.e., over object's relations) and τ translates the refined object into a more abstract modelling language. Sometimes [Kö6] α' could be omitted or replaced by an identity function. ◂

[12] α is a widely adopted symbol field for many different aspects: it is also a relational algebra operator for transitive closures [Agr88], one of the two possible data aggregations presented in [Joh11] or an alignment between two ontologies [ES13].

Since there is an unique function for abstracting different layers, this function also makes possible that each layer could be represented similarly. The possibility of doing so is also confirmed for the previously-introduced "semi-structured data"[13]. For the moment is only sufficient to know that the visual language UML extending the aforementioned model [Obj11, OMG11b], could be expressed with a (semistructured) XML specification, called XMI [OMG11a], through which Data, Model and MetaModel could be all represented. Given that XML can express both data and model layers, XML provides a better abstraction than the former model. Please also note that XML syntax can be also used to represent query languages (XSLT) and, therefore, XML may be also used to represent all data representation layers in the former model. Therefore, semistructured data provide a more uniform representation allowing the definition of α as simply as a XML transformation function.

As it will be outlined in the next chapter, XML has also modelling limitations for representing nested concepts. Therefore, not even XML can be totally adopted as a suitable model for data integration. We procrastinate the discussion for our proposed Data Model to Chapter 5. Bearing this in mind, we're still considering within this chapter MOF as the final data model of choice.

2.2.2 Aggregations

Please observe that this section introduces some running examples that are going to be used along the thesis.

MOF allows to describe α-abstractions between different representation layers but, as previously pointed out, such operator is not allowed to express generalizations inside the same abstraction layer. This consideration implies that other generalization operators working within the same layer (e.g., the Data level) are required. The need of such abstraction mechanism is repeatedly remarked by both data modelling [Pre10, Lar04] and complex systems literature [Joh11], where two distinct types of aggregations are outlined: **part-of** and **is-a**[14]. While the **part-of** relation identifies all the objects that constitute another object within the same abstraction, the **is-a** relation identifies all the objects that could be summarized as one single entity in a coarser representation. Since all those relations are aggregations on the data level, they could be summarized by the following relation, summarizing the information of all the provided objects into one single object [BMM16, BMM17]:

$$\oplus_f : \mathcal{P}(D_O) \to D_O \tag{2.3}$$

Please observe that this definition could be used to arbitrarily aggregate a collection of objects through a transformation (e.g., aggregation) function f. Moreover, $\oplus$ is an object combination operator appearing both `Join` and the `Group By` operators. In the first case, $\oplus$ combines the tuples coming from two distinct relational tables [BMM16], while in the second it aggregates all the objects belonging to the same equivalence class. Within the relational model, we can suppose that $D_R = \varnothing$: now, we can see that both Join and Group By operators are generalizable using the following **nesting operator**:

$$\nu_C^{\oplus_f}(d) = \left\{ \oplus_f(\gamma \cap d) \,\middle|\, \gamma \in C \right\} \tag{2.4}$$

[13] See Chapter 3.2 on page 64 for more details.

[14] In [Joh11], those two operations are ambiguously called α-aggregation and β-aggregation respectively, or even AND-aggregation and OR-aggregation. Both those notations are not-common in both data modelling and in semantic web literature, where the more explanatory **part-of** and **is-a** are used.

where $C \in \mathcal{P}(\mathcal{P}(D_O))$ is a collection of collections, and $\nu^f_C : \mathcal{P}(D_O) \to \mathcal{P}(D_O)$ transforms a collection of objects into a collection of objects. Given this definition, the join operation within the relational model can be defined as:

$$R \bowtie_\theta S = \nu^{\oplus}_{\{\,\{\,a,b \mid b\in S,\theta(a,b)\,\}\mid a\in R\,\}}(R \cup S)$$

This means that we select to merge through the tuple combination function $\oplus$ only the tuples that match the predicate θ. Moreover, the Group By can be expressed as follows:

$$\gamma^f_{A_1,\dots,A_n}(R) = \nu^f_{\{\,\{\,t'\in R \mid t[A_1]=t'[A_1],\dots,t[A_n]=t'[A_n]\,\}\mid t\in R\,\}}(R)$$

This means that we use a generic aggregation function f aggregating the record that share the same common values for the attributes and fields $A_1, \dots, A_n$. Both these operations are used within the relational model for data integration and cleaning purposes [LN07]. Consequently, the $\nu^{\oplus f}_C$ operator can perform both (unary) dimensionality reductions and (n-ary) data operators through combinations.

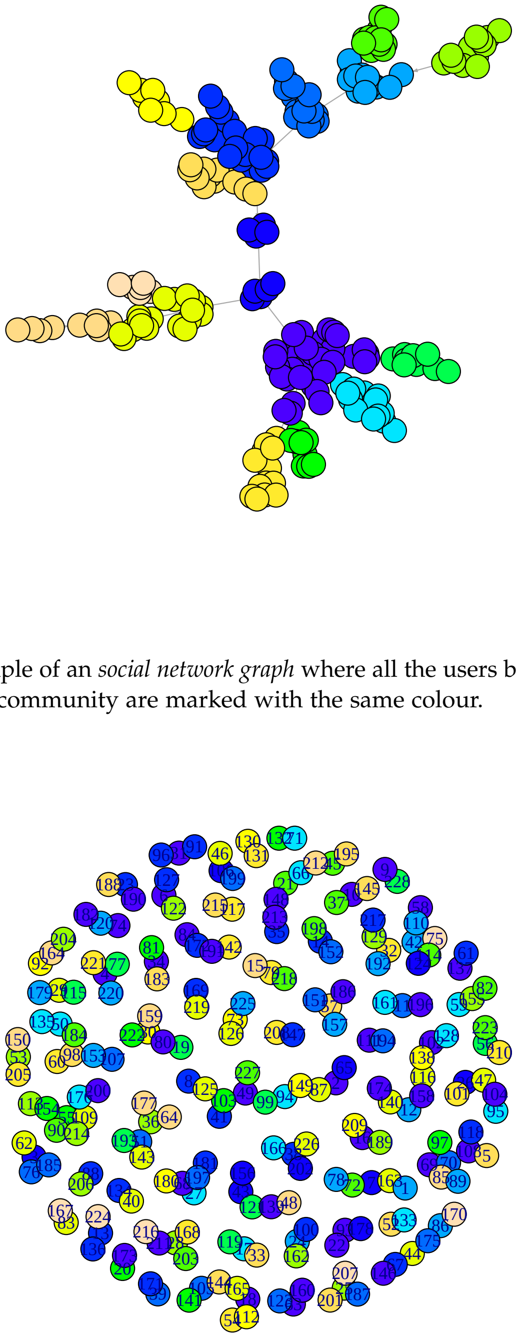
▸ **Example 6.** *The **part-of** aggregation could be used within the context of social network analysis, where all the users that belong to an on-line community are **part-of** such community. Figure 2.15a on the next page provides an example of a social network graph, where each vertex represents an user and each edge represents a friendship relation. Within the social network scenario, we could be also interested in analysing the user activities, such as the messages that have been exchanged between the users. An example of such interaction is provided in Figure 2.15b: please note that from this picture is very hard to guess which is the high level perspective of the messages that have been exchanged among the users and which are the kind of interactions among such communities.*

Figure 2.16b represents the interaction graph where each user replaced by its corresponding community, thus providing a coarser view of the interaction. At this level, the interaction between the communities is more readable and easier to visualize. Similarly, Figure 2.16a represents a simplified view of the social network graph in Figure 2.15a, where the intimacy between social network communities is determined from the friendship relationships between users belonging to different networks.

As we could see from the next example, an operation that only performs an $\nu^{\oplus f}_C$ aggregation is not enough for establishing new links between the connected components, because as a result of the aggregation, we could be also interested in establish relationships between the objects (i.e., vertices), differently from the former example. Hereby, an operation creating new edges between the objects is required.

▸ **Example 7.** *Suppose to have a bipartite graph, where each vertex could be either an author or a paper, and where each edge connects each author to its paper. An example of such graph is provided in Figure 2.17a on page 44. Moreover, on top of this graph, we want to extract an aggregated*[15] *Co-Authorship graph as the one in Figure 2.17b, where each author internally contains a reference to the papers that he has written. Please observe that the rule establishing the relationships between the resulting vertices here differs from the rule required by the former example: in this example we have to establish an edge between the two aggregated vertex vertices if the original vertices are connected by a path of length* 2*, while on the former scenario the path length was* 1*. This suggests that another relevant operation in data integration is the establishment of new relationships among the vertices.*

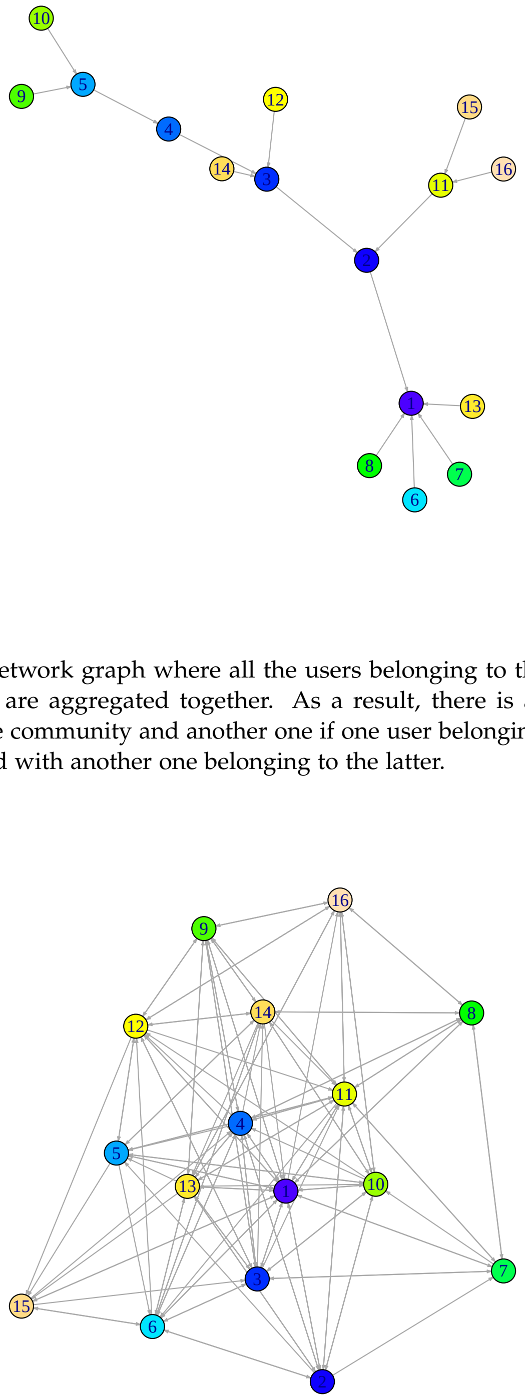
[15] This problem is addressed in [DMR16] and is named **Graph Projection**. Later on on this thesis (Chapter 7), we're going to show that a graph projection operator shall be defined differently, that is by directly providing an implementation of $\nu^{\oplus f}_C$.



(a) An example of an *social network graph* where all the users belonging to the same community are marked with the same colour.

(b) *Interaction graph* between all the users of the users appearing in the previous graph. Each vertex represents an user and an edge represents that the source user has sent a message to a destination user. We use the same community colours as in the social network graph.

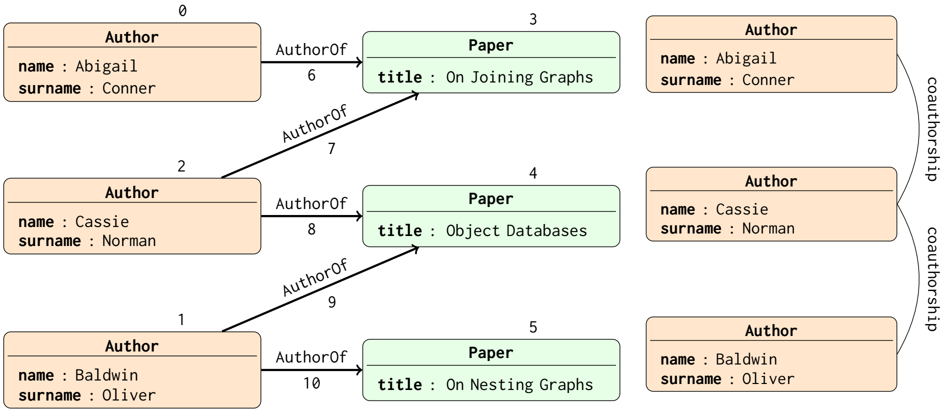
Figure 2.15 Two examples of graph data for social network analysis: a Friendship Graph (2.15a) and an Interaction Graph (2.15b)



(a) Social network graph where all the users belonging to the same community are aggregated together. As a result, there is an edge between one community and another one if one user belonging to the first is friend with another one belonging to the latter.

(b) Upscaling the network interaction at the community level.

Figure 2.16 Coarse grained representation of both the social network graph and the network interaction, aggregated at the community level.



(a) Input bibliographical network.

(b) Aggregated result.

Figure 2.17 Bibliographic network

At this point we want to show why the **is-a** relation could be represented with a nested representation supporting "structural aggregation" instead of a *is-a* edge, as currently done in graph literature. The following example provides an explanation.

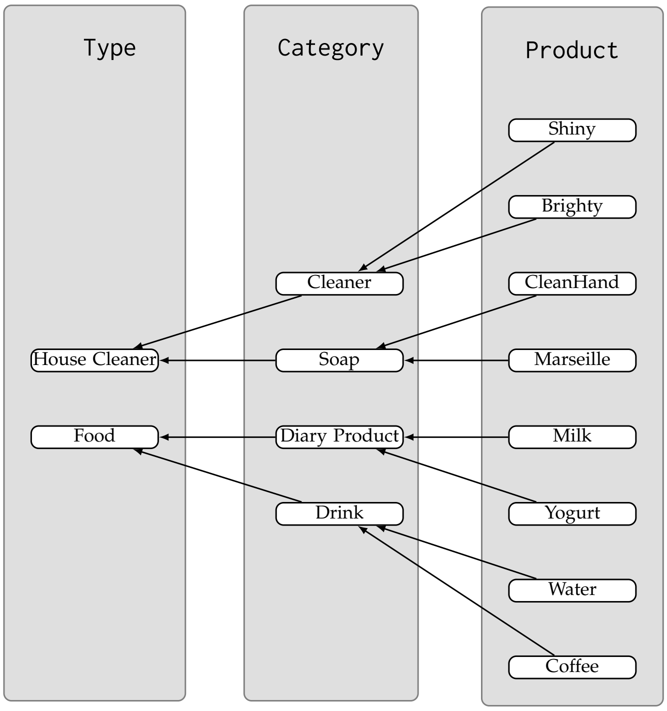
▸ **Example 8.** *Suppose to associate a hierarchy to each product within Figure 2.14 on page 38, thus allowing to represent vertices at different abstraction levels. Hierarchies are "key elements in analytical applications" [VZ14] and graphs allow to provide a direct representation to "non-strict hierarchies". Such graph representation could be also be adopted for transactional data within business process execution [PMB$^+$17], thus allowing to more efficiently represent the data that has to be joined within a standard relational star schema by simply using edge traversals [VTBL13, BDK$^+$13, SFS$^+$15].*

Figure 2.18a provides an example of a hierarchy for some products in Figure 2.14 on page 38 at three different abstraction levels. Even though this representation explicitly expresses the ***is-a*** *relation, it does not provide a good representation of the aggregation. Such aggregation should be represented by the inverse relation of* ***is-a****, that is* ***parent-of****, and such relation should be an n-ary relation containing all its child elements [Joh11]; by doing so we associate value representing the result (e.g., the result of the $v_C^{\oplus f}$ function) to the aggregation of the contained (and aggregated) elements.*

Another standard representation of such hierarchies is provided in Figure 2.18b. In this relational representation each type and category is replicated for each product. This solution is adopted in order to avoid the navigation cost within relational databases through the join queries. This hierarchy representation, on the other hand, is costly for hierarchy updates (data updates and removals require a linear scan of the whole table); also, data aggregations are not feasible within this representation and require an additional join and aggregation cost. On the other hand, this hierarchy representation approach is even worse than the former one, because the notion of the ***is-a*** *relation is completely driven by the "position of the attributes" within the table.*

The nested relational model, on the other hand, provides a better ***parent-of*** *representation by allowing the relations as possible values within a relation's tuple. Figure 2.18c provides an example of how such hierarchy could be modelled within such representation. Even this representation is not beneficial, as within the relational model is not possible to associate to a whole nested relation a data value expressing an aggregated result of its contents.*

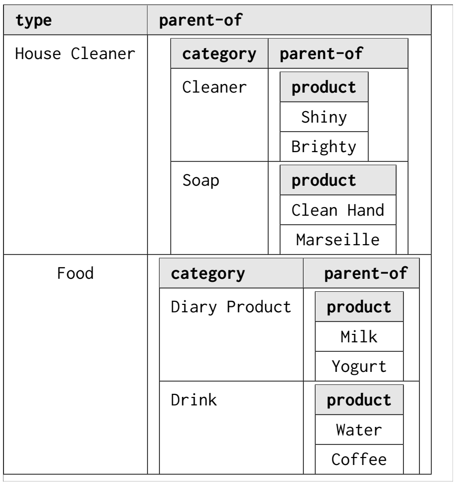
For this reason, the current thesis will provide in Chapter 5 a data model where such part-of



(a) Graph and NOF representation, where each edge represent the **is-a** relation. Each vertex under the same grayed area has the same label of the title of the aforementioned area.

type	category	product
House Cleaner	Cleaner	Shiny
House Cleaner	Cleaner	Brighty
House Cleaner	Soap	Clean Hand
House Cleaner	Soap	Marseille
Food	Diary Product	Milk
Food	Diary Product	Yogurt
Food	Drink	Water
Food	Drink	Coffee

(b) Standard relational representation (tabular) within RDBMS for ROLAP data warehouses. Type and category values are replicated for each product.



<table>
<tr><th>type</th><th>parent-of</th></tr>
<tr><td>House Cleaner</td><td><table><tr><th>category</th><th>parent-of</th></tr><tr><td>Cleaner</td><td><table><tr><th>product</th></tr><tr><td>Shiny</td></tr><tr><td>Brighty</td></tr></table></td></tr><tr><td>Soap</td><td><table><tr><th>product</th></tr><tr><td>Clean Hand</td></tr><tr><td>Marseille</td></tr></table></td></tr></table></td></tr>
<tr><td>Food</td><td><table><tr><th>category</th><th>parent-of</th></tr><tr><td>Diary Product</td><td><table><tr><th>product</th></tr><tr><td>Milk</td></tr><tr><td>Yogurt</td></tr></table></td></tr><tr><td>Drink</td><td><table><tr><th>product</th></tr><tr><td>Water</td></tr><tr><td>Coffee</td></tr></table></td></tr></table></td></tr>
</table>

(c) Standard nested representation of the **is-a** hierarchies in Data Warehouses through the inverse relation, **parent-of**.

■ **Figure 2.18** Different representations for a balanced hierarchy describing the taxonomy associated to the products from Figure 2.14 on page 38. If we assume that each node represents with a different type, we could completely describe such hierarchy at the model level (see Section 2.3.1).

<table>
<tr><th>type</th><th>⊟ quantity</th></tr>
<tr><td>House Cleaner</td><td><table>
<tr><th>category</th><th>⊟ quantity</th></tr>
<tr><td>Cleaner</td><td><table><tr><th>product</th><th>quantity</th></tr><tr><td>Shiny</td><td>1</td></tr></table></td></tr>
<tr><td>Soap</td><td><table><tr><th>product</th><th>quantity</th></tr><tr><td>Marseille</td><td>10</td></tr><tr><td>Marseille</td><td>1</td></tr></table></td></tr>
</table></td></tr>
<tr><td>Food</td><td><table>
<tr><th>category</th><th>⊟ quantity</th></tr>
<tr><td>Diary Product</td><td><table><tr><th>product</th><th>quantity</th></tr><tr><td>Milk</td><td>2</td></tr></table></td></tr>
<tr><td>Drink</td><td><table><tr><th>product</th><th>quantity</th></tr><tr><td>Coffee</td><td>1</td></tr></table></td></tr>
</table></td></tr>
</table>

(a) Completely expanded components. Each quantity referring to a specific `SalesOrder` is associated to its product.

<table>
<tr><th>type</th><th>⊟ quantity</th></tr>
<tr><td>House Cleaner</td><td><table>
<tr><th>category</th><th>⊞ quantity</th></tr>
<tr><td>Cleaner</td><td>1</td></tr>
<tr><td>Soap</td><td>11</td></tr>
</table></td></tr>
<tr><td>Food</td><td><table>
<tr><th>category</th><th>⊞ quantity</th></tr>
<tr><td>Diary Product</td><td>2</td></tr>
<tr><td>Drink</td><td>1</td></tr>
</table></td></tr>
</table>

(b) Aggregating each product, associating to each category the sum of the sold product within the `quantity` field.

<table>
<tr><th>type</th><th>◫ quantity</th></tr>
<tr><td>House Cleaner</td><td><table>
<tr><th>category</th><th>⊞ quantity</th></tr>
<tr><td>Cleaner</td><td>1</td></tr>
<tr><td>Soap</td><td>11</td></tr>
</table></td></tr>
<tr><td>Food</td><td>⊞ 3</td></tr>
</table>

(c) Partially expanded components, where only the Food, Cleaner and Soap components are aggregated.

type	⊞ quantity
House Cleaner	12
Food	3

(d) Completely aggregated components at the type level.

Figure 2.19 Expressing the duality nested relation and aggregated value as provided in modern Data Warehouses [Rol13, PSAH16]. ⊟ represents a fully expanded attribute, ⊞ a completely aggregated attribute, and ◫ a partially aggregated field.

association is possible, as well as expressing relations between the data as in the graph model. An example of how such dualism (contained elements and aggregated value) is possible is introduced in Figure 2.19, where we associate to each product the quantity of the goods sold within each `SalesOrder` *(a). If we consider such aggregation as a "view" over the data, we could think of each aggregated value associated to a nested relation as an expression associated to each of its subcomponents*[16]*. As a consequence, our desired data model should provide both atomic values and expressions composing values and attributes; each* `quantity` *field at both the* `type` *and* `category`

[16] http://rebrand.ly/githuaf109

level should provide both the expression `sum(`$x \mapsto x$`.quantity)` *for the aggregated representation, and a relation over which perform the aforementioned aggregation. Please note that this association could not be provided by the standard nested relational model but, on the other hand, it is frequently used in Data Warehouses for exploring via expansions and collapses the multidimensional components. An example of how to perform such queries is going to be provided later on in Section 6.3.3 on page 182 through the help of some algebraic operators. Moreover, each component could be arbitrarily expanded or aggregated (c): therefore such representation could not fit in the nested relational model. This example will extended and discussed within the nested graph model at page 187, after providing a query* Language *allowing the combination of (nested) graph data alongside with other represented within the same general data model.*

2.3 Multi-database integration

In this section we're interested to attack the **multi-database integration** problem, that can be stated as follows: *given a global schema G of my integrated system and a query q written in a* Language $\mathcal{L}_G$*, evaluate such query among multiple data-sources* $\mathcal{D} = \{ D_1, \dots, D_n \}$ *having different schemas and data representations and provide the final results in a reconciled representation.* This definition is general enough to include both distributed and federated databases data integration.

In order to proficiently solve this problem, we're going to see that ontologies (Section 2.3.1), generalizing data schemas [GP13], attack the schema matching problem (Section 2.3.2) on different data sources with several schemas and representations. Last, we address the problem of evaluating query q on either the original data sources or on the global schema (Section 2.3.3).

2.3.1 Preliminaries: Description Logic and Ontologies

When compared with other data models such as the three world relational model for data mining [CLNP06], MOF also shows its inability to outline and suggest which are the relevant operations to either manipulate the data or to perform assertions on it. Ontologies are now introduced in particular for the latter reason, thus allowing to assert properties over *objects* and *relations* at the Data level, as required by the Model level. Literature provides the following generic definition of an ontology:

▸ **Definition 3.** *An* ***ontology*** *[AH11] is a semantic interpretation of a model establishing relations among model's types and defining generic inference rules. Moreover "an ontology plays a role of a semantic domain in denotational semantics" [SK06] and refers to concepts or entities, that are language and representation independent [HS12].* ◂

Even though the MOF characterization clearly puts the ontology at the Language level, the unclear literature characterization has the problem of distinguishing Models from ontologies [TSL⁺06] because their distinction is "*diverse and frequently contradictory*" [HS12]. On the other hand, other researchers cited in [SK06] claim that the aim of distinguishing Models from ontologies is to separate the representation of the information from the description of the data collected inside it. On the other hand, [SK06] distinguishes the Model from the ontology level by defining a semantic mapping function associating each element of the Model to a collection of the elements of the ontology. Within the MOF characterization, such definition matches with the α function, where each element of the ontology is provided within one single formula predicating the properties associated to the type.

	Description	Syntax $(\cdot)$	Semantics $(\cdot^{\mathcal{I}})$
Roles	atomic role	R	$\{r \in D_R \mid \alpha(r) = R\}$
	inverse role	R^-	$\{(x,y) \mid (y,x) \in R^{\mathcal{I}}\}$
	universal role	U	$D_O \times D_O$
Concepts	atomic concept	A	$\{o \in D_O \mid \alpha(o) = A\}$
	intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
	union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
	complement	$\neg C$	$D_O \backslash C^{\mathcal{I}}$
	top	$\top$	D_O
	bottom	$\bot$	$\varnothing$
	existential restriction	$\exists R.C$	$\{o \in D_O \mid \exists o' \in D_O.(o,o') \in R^{\mathcal{I}} \wedge o' \in C^{\mathcal{I}}\}$
	universal restriction	$\forall R.C$	$\{o \in D_O \mid \forall o' \in D_O.(o,o') \in R^{\mathcal{I}} \Rightarrow o' \in C^{\mathcal{I}}\}$
	at-least restriction	$\geq n\ R.C$	$\left\{ o \in D_O \;\middle\vert\; \left\vert \left\{ o' \in D_O \;\middle\vert\; (o,o') \in R^{\mathcal{I}} \wedge o' \in C^{\mathcal{I}} \right\} \right\vert \geq n \right\}$
	at-most restriction	$\leq n\ R.C$	$\left\{ o \in D_O \;\middle\vert\; \left\vert \left\{ o' \in D_O \;\middle\vert\; (o,o') \in R^{\mathcal{I}} \wedge o' \in C^{\mathcal{I}} \right\} \right\vert \leq n \right\}$
	reflexivity	$\exists R.\textsc{Self}$	$\left\{ x \in D_O \;\middle\vert\; (x,x) \in R^{\mathcal{I}} \right\}$
	nominal	$\{o\}$	$\{o \mid o \in D_O\}$

■ **Table 2.1** Syntax and semantics of SROIQ constructors using Data as a domain for the interpretation under the CWA. SROIQ is the DL language expressing the OWL2.

Among all the possible classes of Language expressing ontologies (such as **frame-based** or **first order logic-based** languages [CGP00]), we chose to describe ontologies using Description Logic (DL) languages. This choice is due to the wide diffusion of OWL, a member of the DL language family, for Semantic Web applications. DL is also widely dealt in current literature, such that their notation is even adopted for expressing generic concepts in data integration literature [ES13]. As a consequence, we provide a DL-biased definition for an ontology:

▸ **Definition 4** (Description Logic Ontology). *Given a set $I = \{\, i_1, \dots, i_n, \dots \}$ of individuals, a* ***description logic ontology*** *O is a pair $(\mathcal{A}, \mathcal{T})$, where both properties of individuals and binary relations between such individuals are expressed.*

In particular, $\mathcal{A}$ is called ***ABox*** *because it contains the following "sorts" of term, called* ***axioms****:*

$$t_{\mathcal{A}} := C(i) \mid R(i,i') \mid C_1 \sqsubseteq C_2 \mid C_1 \equiv C_2 \mid R_1 \sqsubseteq R_2 \mid R_1 \equiv R_2$$

where C describe ***concepts*** *(types) characterizing the individuals, and R represent* ***roles*** *(relationships) among source (i) and destination (i′) individuals. Moreover, inclusions (⊑) and equivalence (≡) between the concepts could be also defined among concepts or roles.*

$\mathcal{T}$ represents the ***TBox*** *containing all the assertions and properties concerning the statements within the ABox. Such assertions are expressed within description logic languages, which expressive power and computational complexity may vary depending on the constructors of choice. An example of such constructors for the SROIQ language is provided in Table 2.1.* ◂

As a consequence, the Model is a part of the ontology (M_O) describing the collection of all the types that could be represented. On the other hand, ABox and TBox statements do not allow to transform individuals into others via transcoding functions ς, because such logic focuses more on expressing properties over existing data than on showing which operations shall be used on top of such data.

Another difference between Description Logic and standard query Language is that the preferred TBox interpretation relies on the "open world assumption" (OWA) [BCM+10]. This implies that the truth value of the DL statements may be true irrespectively of whether or not it is known to be true within the ABox, thus opposing their query interpretation to the closed world assumption (CWA), where only the represented data are assumed to be true, thus describing a "negation as failure" approach (NAF, [RPZ10]). As a consequence, OWA semantics for very expressive languages leads to an undecidable evaluation of the TBox assertions, thus preventing to use such languages for practical interests [BHLS17]. On the other hand, the axiomatic restrictions of the CWA lead to decidable evaluations of DL languages [PSF12], albeit still intractable in some cases. Within the CWA assumption where the Data layer is the domain of the interpretation for both individuals and roles, and assuming that each individual is an actual object of the Model level, $C(i)$ could be interpreted as $\alpha(i) = C$. Please also note that, while MOF allows the representation of a multigraph, thus allowing multiple edges between two vertices, the description logic does not permit to refer to one specific edge among a given source and destination. This last observation makes the standard DL characterization unserviceable on top of our data model, thus requiring to extend such language.

Still within the field of Modal Logics but beyond DL languages[17], the Register Logic language [BFL13] makes constraint checking tractable at the price of the loss of expressive power of graph navigation, but allowing to execute queries always in data polynomial time (NLogspace).

2.3.2 Ontology Alignments and Data Integration

After describing an ontology, we're going to use ontology alignments for attacking the data integration problem. Such class of solutions have received more research acclaim than schema related one [MM09, MM10], which are strictly representation dependent.

At this point we must extend the Description Logic syntax so that individual transcoding functions ς are allowed, in order to be able to express supertypes' transcoding functions. Such transformations will be required in the following scenario in order to say that $C(i) \sqsubseteq C'(\varsigma(i))$.

After describing in the previous sections how ontological alignments are useful within the process of data integration, we can now formally define what an alignment between two ontology is. Moreover, since in some scenarios the alignment uses the data representation to infer such alignment [AGG+15], I extend the usual alignment definition provided for either schema or ontology alignments [ES13, GHKR11] as follows:

▸ **Definition 5** (Ontological Alignment). *An **ontological alignment** $A(O,O')$ of a source ontology O towards a destination ontology O' is a set of tuples called **correspondences**. Each correspondence maps a set of types δt from a model M_O into one type t in $M_{O'}$ using a transcoding function $\varsigma_{\delta t \to t}$. Such correspondence is expressed as $(\delta t, F, t, \varsigma_{\delta t\to t}, s)$, where F is the alignment expressed in description logic axioms between the correspondent types and, whenever $\varsigma_{\delta t\to t}$ is a bijection*[18]*, the inverse function $\varsigma_{t\to\delta t} = \varsigma^{-1}_{\delta t\to t}$ is also provided. An uncertainty score s could be also associated to the accuracy of the alignment [ES13, HGR13].* ◂

[17] DL could be considered as a class of languages providing extensions to modal logic.

[18] If t expresses a **part-of** or an **is-a** relation such as $\delta t \sqsubseteq t$, then sometimes is not possible to unambiguously associate to the aggregation the single disaggregated components.

As a consequence of this definition, given that the ontology subsumes both the informations from the Model (it describes the concepts and roles) and the ones from the MetaModel (TBox expressions), we have that a Language $\mathcal{L}_{MM}$ could be also described by the language expressed in its correspondent ontology. In the case of Description Logic Ontologies, the query Language is the Description Logic itself. Given that both ontologies (containing the schema definition, MM) and queries could always be expressed within the Description Logic ([BCM+10, Chapter 16]), from now on we'll always use the terms *query*, *ontology* and *schema* interchangeably as already did in [Len02].

As outlined by both more recent literature [GHKR11, HGR13] and in the former Example, it is possible to provide alignments between multiple local ontologies into one single global ontology, which in this case it is referred as *hub (ontology)*. Therefore, the previous definition could be extended as follows:

▸ **Definition 6** (Multisource Data Integration System)**.** *A* ***multisource data integration system*** *is defined as a triplet $\langle H, \mathcal{I}, \mathcal{O}\rangle$, where H is the hub ontology, representing the target of all the ontological alignments $A(O,H) \in \mathcal{I}$ having $O \in \mathcal{O}$ as a source (or local) ontology.* ◂

2.3.3 Query Rewriting

In the previous section we focused on performing data integration among different data sources and to reconcile them into one global representation in a data representation independent approach. At this point we should ask at which stage we prefer to execute the query q, either on the still-to-be integrated data sources and then also after the data integration process (Local As a View, [MFK01, NRA+17, SSSF09]), or always at the end of such integration process (Global As a View, [MM06, Luo6, BCC+16]). These two approaches are interchangeable within the data integration task.

▸ **Example 9.** *By rephrasing the data integration approach in the latter subsection, Figure 2.20 provides general data integration framework, where both such approaches could be used. The difference between the two approaches are hidded by the* integration *arrow, that is going to be expanded and analyzed in the following definitions. Our local data either comes with an associated schema (e.g., the SQL dumps containing the constraints on the relational tables) or such schema should be extracted through ad hoc α functions, thus providing an ontology (represented in Figure with blue clique-3 graphs). All such local ontologies must be aligned to the main global hub ontology H, that is either directly provided by the user or obtained "at runtime" from the local ontologies by abstracting over each local schema [BLC+17]. If the data to be integrated contains general purpose concepts such as IBM Watson for Jeopardy [oRD12], then the Ontology Alignment process shall require intermediate general purpose ontologies for producing alignments, such as BabelNet for word relations and DBPedia for more complex and complete data. A more domain specific scenario, such as eHealth, requires more advanced and precise ontologies and taxonomies, such as MeSH and ICD-10 CM, such that the precision of the ontology alignment process can be increased.*

We're now going to see that ontological alignments can be used within a more general data integration framework, that is data representation dependent, and hence we have to introduce functions to translate both data representation and queries. The first class of functions is represented by the class of **translation functions** $\tau_{D_1\to D_2}$, which convert the data representation in D_1 into D_2 in a purely syntactical way:

$$\tau_{D_1\to D_2}\colon D_1 \to D_2 \tag{2.5}$$

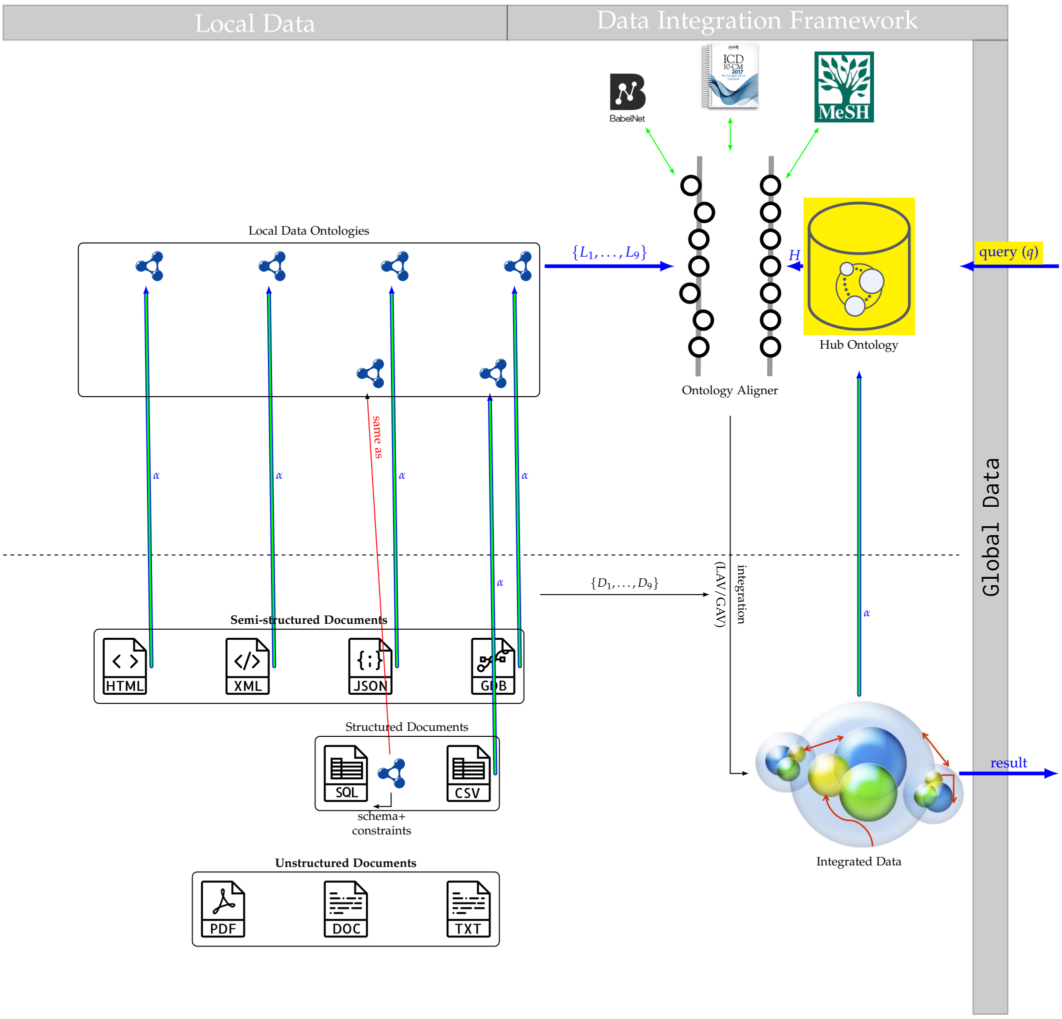


Figure 2.20 Representation of the data integration scenario of both the Local As a View and Global As a View approaches. Beside their definition, the only difference with this representation stands in the representation of the integrated data as either a materialized view of the local sources or as a temporarily query result. The components highlighted in yellow remark the parts that could be directly provided by the external user.

▸ **Example 10.** *The coding of a function allowing to transform a graph representation (Figure 2.17a on page 44) into a JSON one (e.g., Figure 5.7 on page 145) is straightforward, it is commonly used to export data from databases (dump) and does not require to use some specific knowledge concerning either the* Model *or the* Ontology *describing the data. Some data translation function will be provided in Section 5.3 on page 136 for translating any possible data model into the proposed GSM.*

The second operation is the query translation function[19] $Ϙ^{A}_{O_1,O_2}$ from a language $\mathcal{L}_{O_1}$ to another language $\mathcal{L}_{O_2}$ using the information of the alignment $A(O_1,O_2)$ between the two ontologies. More formally:

$$Ϙ\colon \forall O_1,O_2 \in \mathcal{O}.\ A(O_1,O_2) \to \mathcal{L}_{O_1} \to \mathcal{L}_{O_2}$$

[19] It is represented by the qoppa Greek letter: Ϙ, ϙ.

$$\varrho^A_{O_1,O_2} = \varrho(A(O_1,O_2)) \quad s.t. \quad \varrho^A_{O_1,O_2}\colon \mathcal{L}_{O_1} \to \mathcal{L}_{O_2}$$

The Global As a View query rewriting approach, uses the alignments $A(\alpha(D_i),H)$ translating the local schema $\alpha(D_i)$ of database D_i into a global view, thus performing the integration and the schema alignment within the same data representation of H. Using the aforementioned notation, such systems could be summarized with the composition of all the aforementioned functions as in the following definition:

▸ **Definition 7** (Global As a View). *Given a multisource data integration system $\langle H,\mathcal{I},\mathcal{O}\rangle$ for a set of databases $\mathcal{D} = \{D_1,\dots,D_n\}$ having their schemas in $\mathcal{O}$ ($\forall D_i \in \mathcal{D}.\alpha(D_i) \in \mathcal{O}$), a query q could be run on heterogeneous data sources at the end of the data integration steps. First, we translate the data sources into a common representation ($\tau_{\alpha(D_i)\to H}(D_i)$). Given that a schema at the ontology level could even represent a query, such data is now aligned ($\varrho^A_{\alpha(D_i),H}(\alpha(D_i))(\tau_{\alpha(D_i)\to H}(D_i))$): now all the data are in the same representation, and hence they could be aggregated ($\nu_{\cong}(\dots)$) using a clustering algorithm among similar components [SPR17]. Such aggregated data is then queried with q. As a result, we obtain the following expression:*

$$q\left(\nu_{\cong}\left(\varrho^A_{\alpha(D_1),H}(\alpha(D_1))(\tau_{\alpha(D_1)\to H}(D_1)),\dots,\varrho^A_{\alpha(D_n),H}(\alpha(D_n))(\tau_{\alpha(D_1)\to H}(D_n))\right)\right)$$

◂

This definition confirms the intuition expressed in [Len02], stating that GAV systems do not require the translation of the main query q over the different data sources, but only on translating the local schemas into the global one. On the other hand, such systems could not be optimized when q and H are determined on the fly. Nevertheless, this is the traditional approach used in Data Warehouses, where data is extracted, transformed and processed to be compliant to the data warehouse schema expressed through an OLAP query [AGG⁺15]. Moreover, even in traditional data warehouses all the data must be converted into one final schema representation, that is here represented by H [VZ14].

With the second approach, called Local As a View, alignments are used to translate part of the query q to be executed separately on the local sources. Then, the data is translated for the hub schema and integrated as in the previous phase, and then the remaining part of q is run on the global representation.

▸ **Definition 8** (Local As a View). *Given a multisource data integration systems $\langle H,\mathcal{I},\mathcal{O}\rangle$ for a set of databases $\mathcal{D} = \{D_1,\dots,D_n\}$ having their schemas in $\mathcal{O}$, a query q could be run on heterogeneous data sources by first performing a subquery of q over the data that could be queried from D_i accordingly to the alignment A (hence, q gets partially translated in $\varrho^A_{H,\alpha(D_i)}(q)(D_i)$), then the data is necessarily transformed into the global representation $\tau_{\alpha(D_i)\to G}(\varrho^A_{H,L_i}(q)(D_i))$. Later on, the data is integrated into a common data element, over which the remaining part of q, namely q_{END}, is processed to process the remaining part of the query requiring the combination of the pre-process data from the original sources. This process could be sketched as follows:*

$$q_{END}\left(\nu_{\cong}\left(\tau_{\alpha(D_1)\to H}\left(\varrho^A_{H,\alpha(D_1)}(q)(D_1)\right),\dots,\tau_{\alpha(D_n)\to H}\left(\varrho^A_{H,\alpha(D_n)}(q)(D_n)\right)\right)\right)$$

◂

On the other hand, this approach is best suited for data that changes through time, and for which the ontology definition and the data representation could change. Despite the non-negligible cost of query rewriting, this solution provide a best solution when the final data schema could change at any possible query of the database. Nevertheless, this thesis is going to focus on the GAV approach, because it better separates the distinct operations that have to be performed in order to reach our final goal.

2.4 Conclusions

After showing data integration over specific data models, we discussed a general strategy abstracting from particular data representations. We showed that current data models are not able to express structural aggregation, where coarse data representation and finer ones cannot coexist within one single instance. We also showed that current query languages (e.g., SQL and SROIQ) fail at representing either aggregations or alignment tasks. As a consequence, data integration requires both a generalised data model providing the desired structural aggregation and a query language (over such general representation) expressing queries currently used for specific types of data sources. As we're going to see in Chapter 6, our proposed query language is able to express the Ϙ, α and $\nu_{\cong}$ operators for data integration, that can now be only supported outside traditional query languages. All these concepts are going to be addressed and solved within this thesis, either from a formal point of view, or on an algorithmic one (ν and $\bowtie$). With respect to graph data, the present thesis is going to show that it is possible to provide efficient implementations of both graph joins (Chapter 4) and graph nesting (Chapter 7) operators.

The analysis of MetaObject Facility data model showed that query languages should be a part of the desired data model: in particular, a subset of such query languages asserting data properties or transforming data representations should be representable within the data model. Chapter 5 (with special reference to Section 5.1.1 on page 125) will show that this feature allows a straightforward characterisation of structural aggregations within our Generalized Semistructured data Model.

3 Analysing the properties of Data Models and Query Languages

Contents

> *Therefore a science, the advancement of science, and the acquisition of science, is not simply the oblivion of old [scientific] prejudices, or the fall of certain obstacles [to understanding], it is a new grid [of concepts] that masks certain things while allowing for the appearance of new knowledge.*
>
> — Michel Foucault on *The Chomsky-Foucault Debate: On Human Nature*, (1971)

The previous chapter aimed to evaluate present data structures and query languages within the context of data integration. Bearing such considerations in mind, we analyse currently-proposed data models and query languages: as a result, we have that current data models do not support structural aggregation and that current query languages cannot handle both data and schema at the same time. Both these features are required within data integration scenarios. In this chapter, we also compare the previous data model with the property graph data model and its extensions. We draw the following conclusions:

- The relational data model (Section 3.1) does not distinguish entities from relationships (*semantic overloading*), and does not represent data and data properties (M, MM) within the same representation (Section 3.1.1.1).

- Semistructured data provide a multimap association between properties and values (e.g., multiple *tags* with the same name); even if both semistructured and nested relational model allow a content-container relation, only the stream data model meets the requirements for structural aggregation (Section 3.2).
- Current graph data model distinguish entities from relationships but do not provide a structural aggregation over both vertices and edges (Section 3.4).

For each data model, we'll also briefly discuss their query languages and their implications concerning the data manipulation abilities. In particular, we're going to analyse graph query languages' features regarding their underlying data models (Section 3.5).

3.1 Structured data: the Relational Model

A data model is said to be **structured** if it relies on a fixed data model over which some data representation constraints are defined. In particular, a structured data model could be read by a domain-specific program (*query*), which can transform it or create new data expressed in the same model. The most widespread one is the **relational model**, where data are modelled on n-ary mathematical relations r [SS93] to which a *schema* R is associated (denoted as $r(R)$). Such schema constrains the arity of the *tuples* $t \in r$ and the range of values that such tuples could assume. In particular, each schema R is denoted as $R(A_1,\dots,A_n)$, where each A_i is a distinct *attribute* and R is the name given to the relation r. Each relation $r(R)$ is then a subset of the cartesian product $\text{dom}(R) \overset{\text{def}}{=} \text{dom}(A_1)\times\dots\times\text{dom}(A_n)$, where dom is the *domain* function associating a set of possible values to each attribute. Consequently, each tuple may be represented within a single relation only once. In particular, each tuple $t \in r(R)$ is composed by n values[1] $(v_1,\dots,v_n)$ such that $v_i \in \text{dom}(A_i)$ for each[2] $1 \leq i \leq n$. A constraint, called **first normal form (1NF)**, provides some restrictions on the possible domains associated with such attributes: the domain can only map attributes to sets of simple values, thus excluding tuples' values to be either sets of values [Cod71], other relations [EN16] or bag values.

▸ **Example 11.** *Figure 3.1b provides an example of such data model. In particular, each table represents a relation. These are the schema of all the relations in the picture:*

$$\textit{Employee}(\underline{\texttt{id}},\texttt{name},\texttt{surname},\texttt{gender})$$

$$\textit{SalesOrder}(\underline{\texttt{id}},\texttt{date},\texttt{deliveryDate},\texttt{orderer})$$

$$\textit{Product}(\underline{\texttt{id}},\texttt{name},\texttt{category},\texttt{price})$$

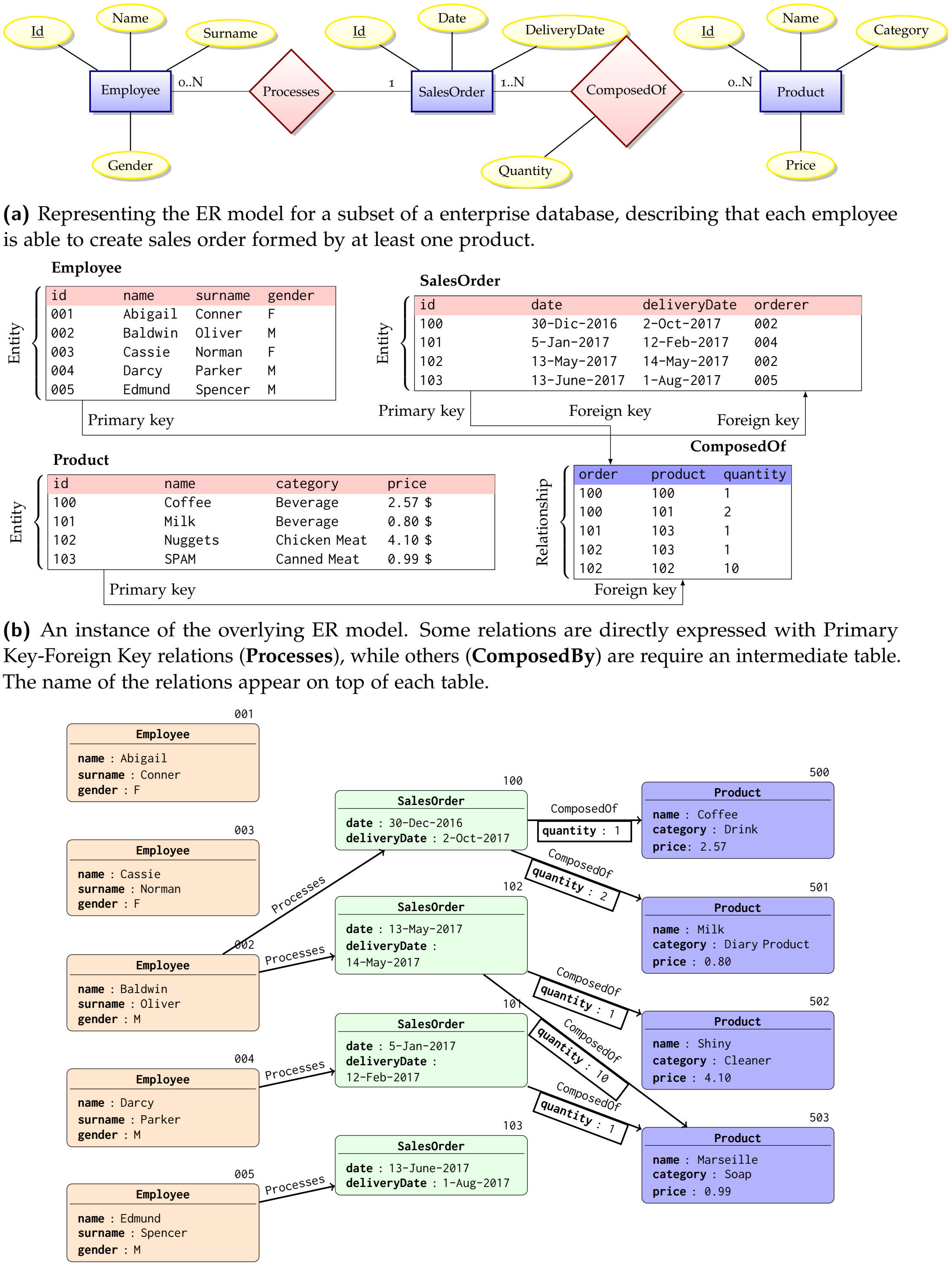
$$\textit{ComposedBy}(\texttt{order},\texttt{product},\texttt{quantity})$$

In particular, each row of each table represents a tuple within the relational model.

3.1.1 Query Languages

Contrarily to what happens for current (graph) query languages, the first language to be developed for this data model was an algebra (called Codd's algebra or Relational Algebra [ACB06, ACPT09, EN16]): it investigated the most elementary operations over relations for

[1] This constraint will be later on called "*horizontal homogeneity*".
[2] This constraint will be later on called "*vertical homogeneity*".



(a) Representing the ER model for a subset of a enterprise database, describing that each employee is able to create sales order formed by at least one product.

Employee

id	name	surname	gender
001	Abigail	Conner	F
002	Baldwin	Oliver	M
003	Cassie	Norman	F
004	Darcy	Parker	M
005	Edmund	Spencer	M

SalesOrder

id	date	deliveryDate	orderer
100	30-Dic-2016	2-Oct-2017	002
101	5-Jan-2017	12-Feb-2017	004
102	13-May-2017	14-May-2017	002
103	13-June-2017	1-Aug-2017	005

Product

id	name	category	price
100	Coffee	Beverage	2.57 $
101	Milk	Beverage	0.80 $
102	Nuggets	Chicken Meat	4.10 $
103	SPAM	Canned Meat	0.99 $

ComposedOf

order	product	quantity
100	100	1
100	101	2
101	103	1
102	103	1
102	102	10

(b) An instance of the overlying ER model. Some relations are directly expressed with Primary Key-Foreign Key relations (**Processes**), while others (**ComposedBy**) are require an intermediate table. The name of the relations appear on top of each table.

(c) Representing the same relational database through a property graph. Please note that this is a faithful representation of the ER schema. The association between vertex and edge id's and their property-value association is described in Section 3.1.2 on page 59 (*Object Identity*)

Figure 3.1 While the process of modelling a relational database (a) requires to distinguish between entities and relationships, its instantiation in a logical database model discards them (b). This information could be preserved within the property graph model (c).

transforming and combining them, and allowed to perform equational reasoning. Such equivalences allow rewriting a relational expression into an equivalent one which is more efficient to compute. Query languages' pre-processing steps use this theoretic result to enhance the query evaluation: declarative query languages such as SQL are compiled into Relational Algebra, over which the aforementioned optimizations can be applied [CG85]. Similar approaches are also used in current column store databases, such as **MonetDB** [3].

Given SQL's characterisation of Section 2.2, we take for granted the basic knowledge of SQL and relational algebra. On the other hand, we now focus on a generalisation of such algebra allowing data mining operations.

3.1.1.1 Data Mining Algebra

Algebraic operators are a useful tools to denote and isolate single relevant data operations over a specific data structure (in this case, the relational model). We shall now discuss which "higher order" operations are relevant in order to meet this goal. As a main reference, we now take the relational data mining algebra proposed in [CLNP06], which main concepts are sketched in Figure 3.6. This algebra suggests that data in D (mainly $D_R = \varnothing$) can represent three distinct concepts: **(i)** the data itself, representing some information content (*data world*, D), **(ii)** the constraints over the data (*intensional world*, I), and **(iii)** the association to each data representation to the constraints that such data satisfies (*extensional world*, E). Within each data world, the traditional set of relational algebra operators extended with a group by Γ and a tuple extension $Calc$ (or their subset) may be used. Such tripartite data model suggests that the relational data model is unable to represent data and its properties within one single representation.

For each data collection c represented in D ($c \in \wp(D)$) we want to remember the data's properties satisfying a given pattern p. Such operation is performed by instantiating such patterns into *regions* in I through the $\kappa_p(c)$ operator.

As a next step, the algebra defines a mining loop operator λ over the regions in I, in which an associated algebraic expression refines the regions until a fixpoint is reached. In particular, the class of the λ operators is the core of a vast range of data mining algorithms, such as frequent (subgraph) mining [JKA⁺17] and (graph) clustering [vDAG12] algorithms.

Then, we want to preserve the data satisfying the mined regions in I by joining them into an intermediate representation in E. The task is achieved through the Pop operator. Given that the relational data model does not provide a uniform data representation for both data and regions, two distinct operations must separate the filtered data (π_A) from the satisfied regions (π_{RDA}).

3.1.2 Representation Problems

Although most of the strengths of the relational data model rely upon a well-established theory [Cod90] that could be easily found in textbooks [GMUW08], this model is no more up with the times. The relational model assumes a centralized structure where all the data is consolidated, and relies on the Closed World Assumption, while modern data analysis techniques may also work on historical data that changes through time. Moreover, *big data* forces companies to distribute data among multiple nodes: therefore it is proved [GL02] that we cannot achieve strong consistency required by the relational model if we want to

[3] https://www.monetdb.org/Documentation/Manuals/MonetDB/Optimizers

achieve high availability and network error tolerance. As a consequence, the model appears to be “static” [BL11] in comparison to other approaches that allow using open world data evolving through time. Those other approaches allow cooperative support [AGG+15] and handle schema mappings changing over time. We are now going to discuss in depth some limitations of the relational model.

Semantic Overloading

Each instance of a relational database can be modelled through the **Entity Relationship (ER)** model [Che76] using graphs; in brief, each real-world entity is defined as a vertex (represented as squared boxes), while the relationships between such entities are modelled as edges (which are represented as rhombus between the vertices). Both entities and relationships can be associated with an attribute (rendered with an oval shape). Figure 3.1a provides a toy example of a usage of such modelling language, where some employees can process sale orders within a company.

As a next step, we have to transform this representation into relational tables: as we could see from Figure 3.1b, such model is table based, and hence it does not distinguish entities from relationships. Consequently, this whole design approach suffers from **semantic overloading**. Moreover, some relationships have to be expressed through **referential integrity constraints**, and occasionally represent relationships through additional tables. In particular, referential integrity constraints require that every value of a *foreign key* must exist as a value of a *primary key* within the referred table.

On the other hand, graphs solve the problem of the semantic overloading by representing the entities as vertices and relationships as edges as in Figure 3.1c. Despite this, such data model still has to implement some integrity constraints for consistency checking (e.g. when a vertex is removed, all the incoming and outgoing edges to (and from) that vertex must be removed).

Data Homogeneity

Within the relational model, each tuple could be only represented within a relation and has no independent identification or existence outside a relation. All the relations’ tuples must share the same attributes (**horizontal homogeneity**) which must always contain values from the same domain (**vertical homogeneity**). As a consequence, relational databases do not cope with data having a flexible and schemaless representation.

On the other hand, the property graph model provides an ideal representation, because it does not force to represent vertices and edges into a homogeneous schema [VTBL13]. As shown in Figure 3.1c, each vertex and edge make explicit the entity from which it comes from through its label. Moreover, no specific constraint as the relations’ schema is specified for both vertices and edges, thus achieving horizontal and vertical heterogeneity.

Object Identity

The original relational model does not formalise explicit constraints such as *primary keys* identifying each tuple within a relation through a (set of) attribute(s). Those constraints can be expressed within Relational DataBase Management Systems (RDBMSs), thus extending the theoretical relational model. In this scenario, even if the user-defined primary key could be associated with a specific column within a table, the same key value could be still used to identify two different relational table’s tuples *when no specific constraints are* *specified*. As an example, Figure 3.1b shows that `SalesOrder`'s id and `Product`'s id share the same primary key values, even if they refer to different instances of entities. This phenomenon does not arise in other graph data models (e.g. RDF [AH11], EPGM [JPT+16] and Networks [Joh11]), because each node is associated with a unique identifier [GHMP11], while the same problem affects graph databases (e.g., in Neo4J [HG16]), where vertex and edge ids are used to index the graph elements and not to unambiguously identify them.

The Object Identity problem has already been formulated and solved within the Object-Oriented data model: for each object representing an instance of a relation $\mathfrak{R}$, the unique identifier could be uniquely determined via a function $f_{\mathfrak{R}}$ (called *Skolem Functor*) which computes the unique id from the object's terms [Cab98]. This function generates new IDs for each newly created objects. Consequently, such objects could be easily implemented in current programming languages (e.g. Java) where both hashing functions and equivalence predicates are provided for each class. On the other hand, this approach does not allow to establish explicit primary keys and makes hard to retrieve the stored information. Still, the vertex and edge identifier solution could be seen as a relaxation of this Object Identification, and hence it could be adopted by the property graph such that one single set of attributes and values (and even labels) corresponds to that id.

Recursive queries

An early extension of the relational algebra [AU79] tried to introduce an algebraic operator providing a transitive closure over the directed binary relations stored in relational tables via fixpoint evaluation. As we could see from the previous discussions, the implementation of object identifiers is the main prerequisite for checking whether the tuples have been already visited or not. This fix-point operator would be later named α [Agr88]. Recursive queries became standard with **SQL:1999**, where a `WITH RECURSIVE` was added to the SQL syntax, and a least fix-point semantic was associated with this clause, by assuming each distinct tuple as a different object.

Generalization and Inheritance

Another feature of the Entity Relationship model is the ability of expressing *generalizations* (**is-a**), through which we state that an entity is derived from another one (similarly to subclassing in object oriented programming). Such generalizations are modelled within the ER model through `ISA` triangular nodes as the one in Figure 3.2a on the next page, extending the previous ER diagram in Figure 3.1a. In particular, we want to distinguish two different kind of `Employees`, the `Vendors` and the `Managers`, having different tasks: the `Managers` administer the `Vendors`, which could process the `SalesOrders`. Figure 3.2b show that there could be ER model instantiations in the relational model that remove such explicit information for efficiency reasons, as explained in the following example:

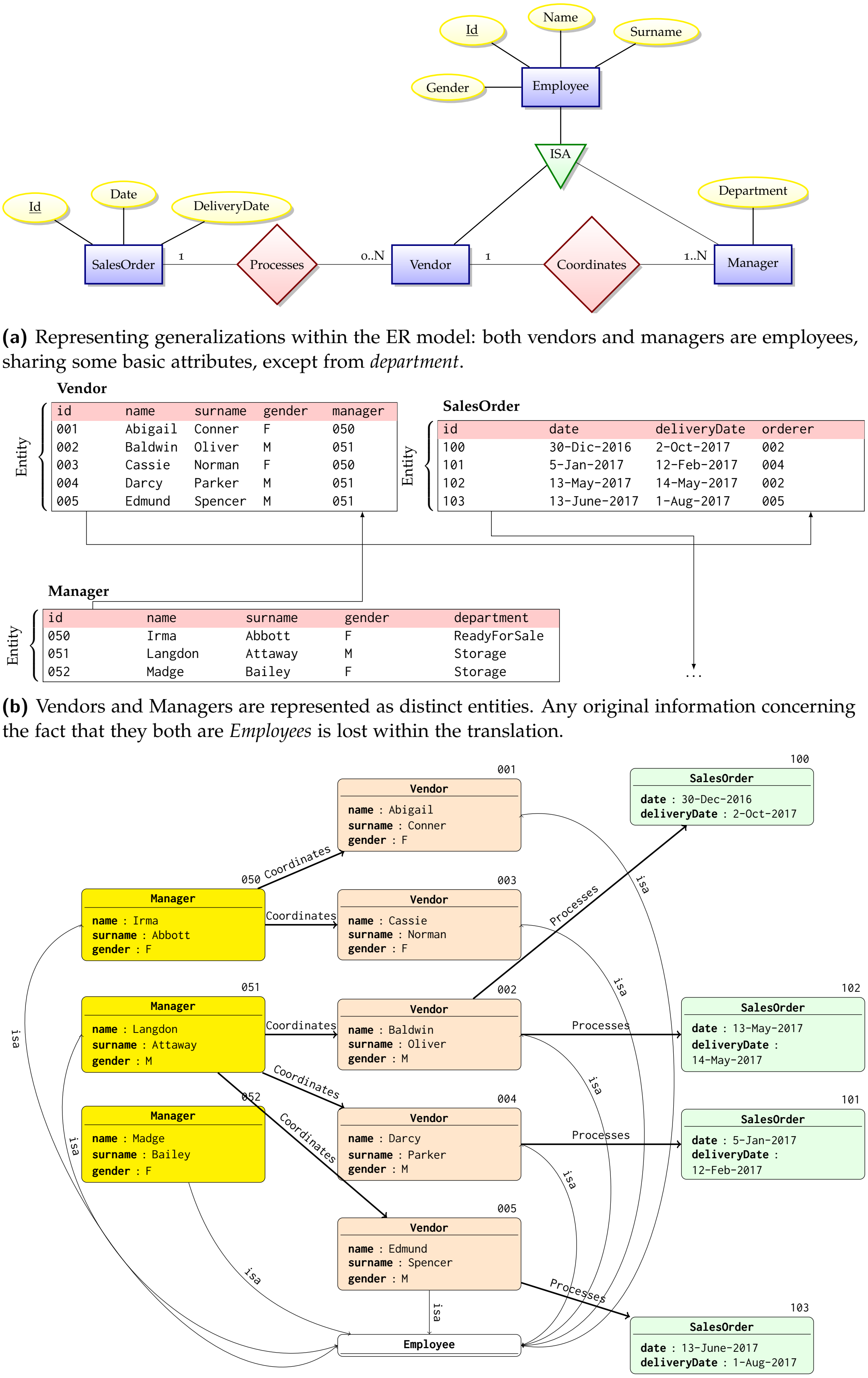
▸ **Example 12.** *Instead of using just two distinct relations as in Figure 3.2b,* `Vendor` *and* `Manager`*, with the following schemas:*

$$Vendor(\underline{\texttt{id}},\texttt{name},\texttt{surname},\texttt{gender},\texttt{manager})$$

$$Manager(\underline{\texttt{id}},\texttt{name},\texttt{surname},\texttt{gender},\texttt{department})$$

we could have used the following schema, preserving the `isa` *generalization:*

$$Employee(\underline{\texttt{id}},\texttt{name},\texttt{surname},\texttt{gender})$$



(a) Representing generalizations within the ER model: both vendors and managers are employees, sharing some basic attributes, except from *department*.

Vendor

id	name	surname	gender	manager
001	Abigail	Conner	F	050
002	Baldwin	Oliver	M	051
003	Cassie	Norman	F	050
004	Darcy	Parker	M	051
005	Edmund	Spencer	M	051

SalesOrder

id	date	deliveryDate	orderer
100	30-Dic-2016	2-Oct-2017	002
101	5-Jan-2017	12-Feb-2017	004
102	13-May-2017	14-May-2017	002
103	13-June-2017	1-Aug-2017	005

Manager

id	name	surname	gender	department
050	Irma	Abbott	F	ReadyForSale
051	Langdon	Attaway	M	Storage
052	Madge	Bailey	F	Storage

(b) Vendors and Managers are represented as distinct entities. Any original information concerning the fact that they both are *Employees* is lost within the translation.

(c) Representing the generalization within the graph data model through `isa` edges.

■ **Figure 3.2** Comparing the ability of expressing generalization between the relational model and property graphs.

$$\mathit{Vendor}(\texttt{employee\_id},\texttt{manager}) \quad \mathit{Manager}(\texttt{employee\_id},\texttt{department})$$

This representation comes with the following computational price: if we want to retrieve the personal informations' for each `Vendor` *and* `Manager`*, we must always perform join queries while, in the previous cases, we could simply access the* `Vendor` *and the* `Manager` *tables using the following SQL queries:*

```
SELECT id, name, surname, gender, manager
FROM Employee, Vendor
WHERE id = employee_id

SELECT id, name, surname, gender, department
FROM Employee, Manager
WHERE id = employee_id
```

Such SQL queries could be respectively expressed in Relational Algebra as follows:

$$\pi_{\texttt{id,name,surname,gender,manager}}(\mathit{Employee} \bowtie_{\texttt{id=employee\_id}} \mathit{Vendor})$$

$$\pi_{\texttt{id,name,surname,gender,department}}(\mathit{Employee} \bowtie_{\texttt{id=employee\_id}} \mathit{Manager})$$

On the other hand, graph databases maintain the `isa` relation as metadata [LS99] information, because both data and metadata could be expressed within the same graph database instance[4] [VTBL13]. Figure 3.2c shows how to associate metadata (the `Employee` information with the `isa` edges) within the original graph data, so that the `isa` information could be traversed only when required. It is also showed that accessing to such information through graph traversal queries (*path joins*) is more efficient than traversing relations through relational joins. As an example, within a graph database we could retrieve the subgraph containing all the Vendors with their Managers with the following Cypher query:

```
MATCH (vendor:Vendor)-[:isa]->(:Employee)<-[:isa]-(boss:Manager)
MATCH path = (vendor:Vendor)-[:hasManager]->(boss:Manager)
RETURN path
```

After rewriting the Property Graph model into the RDF graph model as described in [DSP+14] (see Section 3.4 on page 73), we could express the same query in SPARQL as follows:

```
PREFIX company: <http://company.com/graphdb#>
CONSTRUCT {
        ?vendorid company:name ?vname;
                  company:surname ?vsurname;
                  company:gender ?vgender;
                  company:hasManager ?managerid.
        ?managerid company:name ?mname;
                   company:surname ?msurname;
                   company:gender ?mgender;
                   company:department ?dept.
} WHERE {
```

[4] This property will be also later on important for expressing data integration tasks.

```
    company:Vendor a company:Employee.
    ?vendorid a company:Vendor;
                  company:name ?vname;
                  company:surname ?vsurname;
                  company:gender ?vgender;
                  company:hasManager ?managerid.
    company:Manager a company:Employee.
    ?managerid a company:Manager;
                   company:name ?mname;
                   company:surname ?msurname;
                   company:gender ?mgender;
                   company:department ?dept.
}
```

Graph query languages such as Cypher and SPARQL are going to be described in Section 3.5 on page 76.

3.1.3 Representing graphs

The previous sections showed how relational databases could be completely described by Property Graphs [HG16]. We could formalize the data structure used in Figure 3.1c as follows:

▸ **Definition 9** (Property Graph). *A* ***property graph*** *is a tuple* $(V,E,L,A,U,\ell,\kappa,\lambda)$*, such that* V *and* E *are sets of distinct integer identifiers* ($V \subseteq \mathbb{N}$, $E \subseteq \mathbb{N}$, $V \cap E = \varnothing$)*.* L *is a set of labels,* A *is a set of attributes and* U *a set of values.*

Concerning the functions, $\ell\colon V \cup E \to \mathcal{P}(L)$ *is a function associating to each vertex and edge of the graph a set of labels*[5]*;* $\kappa\colon V \cup E \to A \to U$ *is a function associating, for each vertex and edge within the graph and for each attribute within* A*, a value in* U*; last,* $\lambda\colon E \to V \times V$ *is the function associating to each edge* $e \in E$ *a pair of vertices* $\lambda(e) = (s,t) \in V \times V$*, where* s *is the source vertex and* t *is the target.* ◂

Given the data model provided by this definition, we can now instantiate the graph illustrated in Figure 3.1c within this very model as shown by the following example:

▸ **Example 13.** *Graph in Figure 3.1c is described by the following vertex set:*

$$V = \{\texttt{001},\texttt{002},\texttt{003},\texttt{004},\texttt{005},\texttt{100},\texttt{101},\texttt{102},\texttt{103},\texttt{500},\texttt{501},\texttt{502},\texttt{503}\}$$

The label association is defined as follows:

$$\ell(\texttt{001}) = \ell(\texttt{002}) = \ell(\texttt{003}) = \ell(\texttt{004}) = \ell(\texttt{005}) = \{\textbf{Employee}\}$$

$$\ell(\texttt{100}) = \ell(\texttt{101}) = \ell(\texttt{102}) = \ell(\texttt{103}) = \{\textbf{SalesOrder}\}$$

$$\ell(\texttt{500}) = \ell(\texttt{501}) = \ell(\texttt{502}) = \ell(\texttt{503}) = \{\textbf{Product}\}$$

Moreover, we could model the edges as follows:

$$E = \{\texttt{800},\texttt{801},\texttt{802},\texttt{803},\texttt{804},\texttt{805},\texttt{806},\texttt{807},\texttt{808}\}$$

[5] I prefer to associate to each vertex and edge a set of labels instead of one single label in order to be able to express the Neo4J [Neo13, HG16] each vertex and edge could have more than one possible label

Even the edges could show labels as follows:

$$\ell(800) = \ell(801) = \ell(802) = \ell(803 = \{\mathbf{Processes}\}$$

$$\ell(804) = \ell(805) = \ell(806) = \ell(807 = \ell(808 = \{\mathbf{ComposedOf}\}$$

In this scenario, edges do not have associated attributes or values, and hence could be modelled as relational tuples with 0 *arity. Edges' sources and destinations are modelled through the* λ *function:*

$$\lambda(800) = (002,100) \quad \lambda(801) = (002,102) \quad \lambda(803) = (004,101) \quad \lambda(804) = (005,103)\dots$$

In particular, the properties and the values can be associated with both vertices and edges. For example, Employee *Abigail Conner is modelled as follows:*

$$\kappa(001,\texttt{name}) = \mathit{Abigail} \quad \kappa(001,\texttt{surname}) = \mathit{Conner} \quad \kappa(001,\texttt{gender}) = F$$

while the association between Coffee and the first SalesOrder *in chronological order is modelled with the following properties:*

$$\kappa(804,\texttt{quantity}) = 1$$

3.2 Nested Relational Model, Semistructured data and Streams

A data model is said to be **semistructured** if it relies on a fixed data model over which **no** constraints are imposed. Given this definition, we could now distinguish the **nested relational model** from the **XML data model**: even though both models rely on the relaxation of the 1NF by allowing nested data representations as possible data values (thus allowing the representation of hierarchical data), the former model still requires the nested relations to be compliant to a given schema, while the latter does not. As a consequence, the first model still suffers from the "data homogeneity" problem addressed for the Relational Model in Section 3.1.2 on page 59, while the second does not.

▸ **Example 14.** *As we could see in Figure 3.3, XML achieves the 1NF relaxation through the definition of tags: a tag is a construct beginning with the character "<" and ending with ">": all the data is contained between the start tag (e.g., "*<Database>*") and the end tag (e.g., "*</Database>*"), that could cointain either textual elements or other tags. To each tag, some properties and values could be associated (e.g. the tag "*<Vendor id="001">. . . </Vendor>*" contains a property* id *with value* 001*). An empty tag (e.g., "*<salesOrder />*") represents a tag containing no information. Table 3.1 provides the same representation for nested relations: in this case the nested data representation is represented by relations that could be used as valid values for some attributes (e.g.,* salesOrder*), provided that the relation has a fixed schema. The empty tag could be represented by either a* NULL *value or an empty relation.*

A graphical representation of such tree is provided in Figure 3.4 in the footsteps of [MM06]. Please note that such alternative representation does not consider the tag's attributes like "id", and only provides label-less edges. A more complete XML data model representation in this regard is provided by Lu [Lu06], also allowing to integrate relational data with XML and Object Oriented databases.

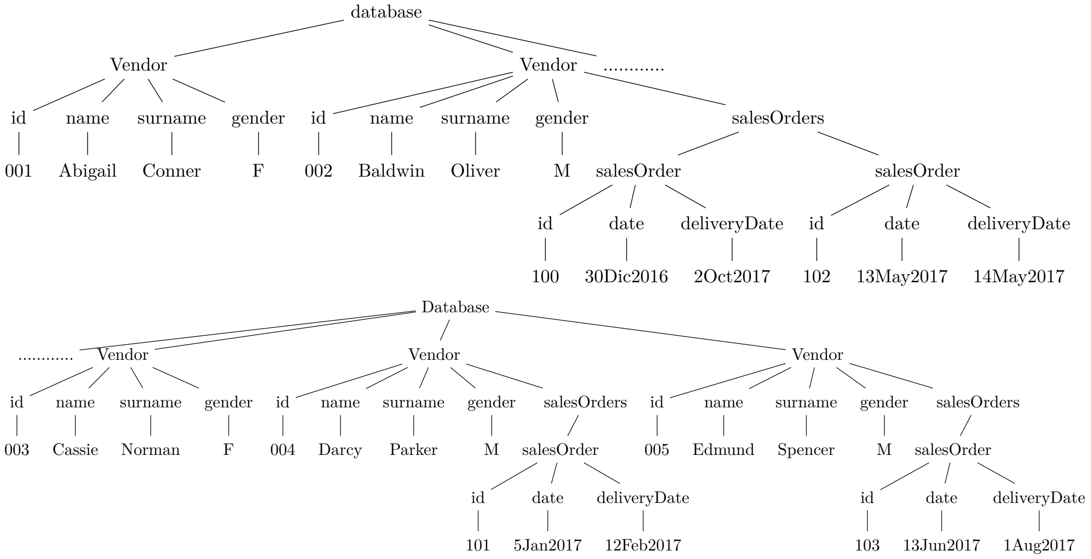
The flexibility of the XML representation is also evident by the fact that one single tag can contain multiple tags with the same name, while the nested relational model only allows one single attribute per nested relation. As a consequence, the XML model generalizes the Object model represented in Definition 1 on page 36 because within that

```
<Database>
        <Vendor id="001">
                <name>Abigail</name>
                <surname>Conner</surname>
                <gender>F</gender>
                <salesOrder />
        </Vendor>
        <Vendor id="002">
                <name>Baldwin</name>
                <surname>Oliver</surname>
                <gender>M</gender>
                <salesOrders>
                        <saleOrder id="100">
                                <date>30-Dic-2016</date>
                                <deliveryDate>2-Oct-2017</deliveryDate>
                        </saleOrder>
                        <saleOrder id="102">
                                <date>13-May-2017</date>
                                <deliveryDate>14-May-2017</deliveryDate>
                        </saleOrder>
                </salesOrders>
        </Vendor>
        <Vendor id="003">
                <name>Cassie</name>
                <surname>Norman</surname>
                <gender>F</gender>
                <salesOrder />
        </Vendor>
        <Vendor id="004">
                <name>Darcy</name>
                <surname>Parker</surname>
                <gender>M</gender>
                <salesOrders>
                        <saleOrder id="101">
                                <date>5-Jan-2017</date>
                                <deliveryDate>12-Feb-2017</deliveryDate>
                        </saleOrder>
                </salesOrders>
        </Vendor>
        <Vendor id="005">
                <name>Edmund</name>
                <surname>Spencer</surname>
                <gender>M</gender>
                <salesOrders>
                        <saleOrder id="103">
                                <date>13-Jun-2017</date>
                                <deliveryDate>1-Aug-2017</deliveryDate>
                        </saleOrder>
                </salesOrders>
        </Vendor>
</Database>
```

Figure 3.3 Representing the association between the `Vendors` and their `SalesOrders` in Figure 3.1b with an XML representation.

<table>
<tr><th>id</th><th>name</th><th>surname</th><th>gender</th><th>salesOrders(id,date,deliveryDate)</th></tr>
<tr><td>001</td><td>Abigail</td><td>Conner</td><td>F</td><td></td></tr>
<tr><td>002</td><td>Baldwin</td><td>Oliver</td><td>M</td><td><table><tr><th>id</th><th>date</th><th>deliveryDate</th></tr><tr><td>100</td><td>30-Dic-2016</td><td>2-Oct-2017</td></tr><tr><td>102</td><td>13-May-2017</td><td>14-May-2017</td></tr></table></td></tr>
<tr><td>003</td><td>Cassie</td><td>Norman</td><td>F</td><td></td></tr>
<tr><td>004</td><td>Darcy</td><td>Parker</td><td>M</td><td><table><tr><th>id</th><th>date</th><th>deliveryDate</th></tr><tr><td>101</td><td>5-Jan-2017</td><td>12-Feb-2017</td></tr></table></td></tr>
<tr><td>005</td><td>Edmund</td><td>Spencer</td><td>M</td><td><table><tr><th>id</th><th>date</th><th>deliveryDate</th></tr><tr><td>103</td><td>13-Jun-2017</td><td>1-Aug-2017</td></tr></table></td></tr>
</table>

■ **Table 3.1** Representing the association between the Vendors and their SalesOrders in Figure 3.1b with a nested tabular representation. Each attribute containing a nested relation exposes both the name of the attribute and the schema associated to the relations that it contains.



■ **Figure 3.4** Representing the association between the Vendors and their SalesOrders with an tree representation of Figure 3.3. Please note that the two trees represent the same XML document, that has been here split in two parts due to page size and readability limitations. The representation of choice is the one provided in [MM06], where no tag attributes are considered. The XML's empty tags are removed since they are not supported by the data model.

definition, the attribute-value association is implicitly represented by a map while, in this case, such association can be represented with a multimap.

Data models having no fixed schema could be always associated to a schema in a later step: for example, tag schema could be externally validated through **XML Schema**[6] [Vli02], expressing such constraints within the same XML data representation. As a result, while the XML data model is widely adopted due to its flexibility in web technologies, the nested

[6] Other languages that could be used to express XML schemas are RelaxNG and DTD, even though such languages do not use the XML syntax.

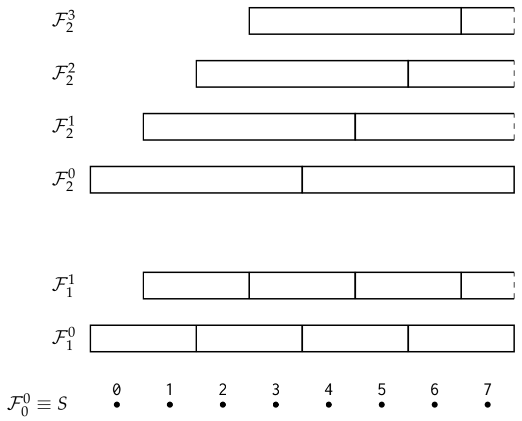


Figure 3.5 A representation of an *incremental feature extraction* scenario, where the window size is $w = 2$ and the update rate is $T = 1$. If we use α as the sum function, then $\mathcal{F}_1^0[0] = 1$ and $\mathcal{F}_2^0[0] = 6$, where $\mathcal{F}_2^0[0] = \mathcal{F}_1^0[0] + \mathcal{F}_1^0[1] = (0+1) + (2+3)$.

relational model is not, even though it is adopted for providing multidimensional view of structured data (thus, homogeneous) as in Pentaho [Rol13] and JasperServer [PSAH16]. The reason why the nested relations are preferred for visualizing nested multidimensional data is evident by comparing the XML representation in Figure 3.3 with the nested representation in Table 3.1. XML is more verbose and less compact and intuitive than the nested relational tables.

Last, we must also observe that the previous nested relational models and semistructured models do not allow a complete representation of the **stream data model**. This data model is an extension of collection representations [LWZ04, AW04], where the incoming stream $S[0,\dots,n-1]$ is represented as a collection of n values. On top of such stream, we may create features $\mathcal{F}_{j+1}^T$ representing data at a given resolution j, which it may be sampled after a (possibly zero) update rate T [BS07]. Each value $\mathcal{F}_{j+1}^T[i] \in \mathcal{F}_{j+1}^T$ may be computed from a finer data resolution $\alpha(\mathcal{F}_j^{T \bmod w^j}[iw,\dots,(i+1)w-1])$, where α is an abstraction function [Joh11] allowing dimensionality reduction for acquiring higher level information [BS07].

▸ **Example 15.** *Figure 3.5 provides an example of a data stream where α is the sum function over naturals. This implies that each possible sub-collection $\mathcal{F}_j^{T \bmod w^j}[iw,\dots,(i+1)w-1] \subset \mathcal{F}_j^{T \bmod w^j}$ must both represent a value and may also remember the coarser data levels, too. In particular, such feature is missing in current nested and document based data models, thus making impossible the definition of structural aggregations as required by current MOLAP systems (see Example 8 on page 44).*

Last, please observe that each data element can be part of different multiple stream representations. For example, both raw data and aggregated data are contained in multiple coarser representations. In the first case, $S[1] = 1$ is contained in both $\mathcal{F}_1^0[0]$ and $\mathcal{F}_1^1[0]$; in the second case, $\mathcal{F}_1^0[1]$ is contained in both $\mathcal{F}_2^0[0]$ and $\mathcal{F}_2^2[0]$.

3.2.1 Query languages

It has been already proved that any nested relational query expressed in the nested calculus could be mapped into a relational algebra query run over a flattened representation of the nested relational algebra [PG92]. As a result, such nested calculus benefits from the same rewriting rules presented within the relational model, and hence benefits from the same query optimizations. Since the relational algebra has been also extended to XML trees [Lu06] we have that even such algebras take advantage of such rewriting rules [MM06]. Despite the interest of data integration between structured and semi-structured that such query language made feasible, none of the aforementioned algebras have been implemented yet in real systems.

On the other hand, traversing query languages over one single representation and over semistructured data have been more successful: in particular, **XPath** [BBC$^+$15] is very used within other semistructured query languages, such as **XQuery** [Wal07] and **XSLT** [Tid08], which transform semistructured documents to other documents in any other format. In the footsteps of [MM06], we're going to analyse a minimal subset of XPath, merely concerning the path traversal, thus discarding all the axis notation and the selection predicates.

▸ **Definition 10** (Minimal XPath). *A minimal XPath expression* <μXPath> *is formed by a concatenation of traversal operator* <c>*, that can be either* / *(direct descendant) or* // *(any node contained at any depth) to which a collection selector (*<s>*) may be applied (e.g.* /$^{\text{``Tag''}}$*, which means to select the nodes directly descendant from root contained in the* Tags *collection).*

$$\texttt{<s>} := \varepsilon \mid \star \mid \texttt{[name]}$$

$$\texttt{<c>} := /^{\texttt{<s>}} \mid //^{\star}$$

$$\texttt{<}\mu\texttt{XPath>} := \texttt{<c><s>} \mid \texttt{<c><s><}\mu\texttt{XPath>}$$

◂

The semantics associated to such query language [Wad00] returns the forest (tree collection) that is reached after the stepwise evaluation of the XPath expressions.

3.2.2 Representation problems

Given that property graphs do not relax 1NF, they do not allow nested representations. Therefore, we cannot say that they provide a better representation than the data models that have been now presented, but we can only say that are complementary models. As a consequence, this section is going to be later on used as a set of requirements that general semistructured data models must satisfy for solving the representation problems. The resulting data model will be presented in a separate chapter (Chapter 3.4 on page 73).

Semantic Overloading

Both models suffer from semantic overloading: while the motivation for the nested relational model could be found in Section 3.1.2 on page 59, the XML model suffers from this problem because it uses tags for expressing entities (e.g., Vendor), properties (e.g., name, surname) to which values could be associated, and nesting components (e.g., salesOrders). This problem is practically solved within the proposed property graph model, where labels $l \in L$ belong to a different set than attributes $a \in A$, and attributes could be not expressed in two possible different ways (e.g. tag attributes and tags containing values). On the other

<table>
<tr><th>id</th><th>name</th><th>surname</th><th>gender</th><th>salesOrders(. . .)</th></tr>
<tr><td>001</td><td>Abigail</td><td>Conner</td><td>F</td><td></td></tr>
<tr><td>002</td><td>Baldwin</td><td>Oliver</td><td>M</td><td>
<table>
<tr><th>id</th><th>date</th><th>deliveryDate</th><th>products(. . .)</th></tr>
<tr><td>100</td><td>30-Dic-2016</td><td>2-Oct-2017</td><td>
<table>
<tr><th>id</th><th>quantity</th><th>name</th><th>category</th><th>price</th></tr>
<tr><td>100</td><td>1</td><td>Coffee</td><td>Drink</td><td>2.57</td></tr>
<tr><td>101</td><td>2</td><td>Milk</td><td>Diary Product</td><td>0.80</td></tr>
</table>
</td></tr>
<tr><td>102</td><td>13-May-2017</td><td>14-May-2017</td><td>
<table>
<tr><th>id</th><th>quantity</th><th>name</th><th>category</th><th>price</th></tr>
<tr><td>102</td><td>1</td><td>Shiny</td><td>Cleaner</td><td>4.10</td></tr>
<tr><td>103</td><td>10</td><td>Marseille</td><td>Soap</td><td>0.99</td></tr>
</table>
</td></tr>
</table>
</td></tr>
<tr><td>003</td><td>Cassie</td><td>Norman</td><td>F</td><td></td></tr>
<tr><td>004</td><td>Darcy</td><td>Parker</td><td>M</td><td>
<table>
<tr><th>id</th><th>date</th><th>deliveryDate</th><th>products(. . .)</th></tr>
<tr><td>101</td><td>5-Jan-2017</td><td>12-Feb-2017</td><td>
<table>
<tr><th>id</th><th>quantity</th><th>name</th><th>category</th><th>price</th></tr>
<tr><td>103</td><td>1</td><td>Marseille</td><td>Soap</td><td>2.57</td></tr>
</table>
</td></tr>
</table>
</td></tr>
<tr><td>005</td><td>Edmund</td><td>Spencer</td><td>M</td><td>
<table>
<tr><th>id</th><th>date</th><th>deliveryDate</th><th>products(. . .)</th></tr>
<tr><td>103</td><td>13-Jun-2017</td><td>1-Aug-2017</td><td></td></tr>
</table>
</td></tr>
</table>

Table 3.2 Extension of the nested representation of Table 3.1 showing the deficiencies of such data representation over multiple nesting levels.

hand, the nested extension of the property graph model should allow to contain vertices and edges, as nested relations could contain other relations as values, as well as XML tags' contents could be other XML tags.

Data Homogeneity

As previously discussed, only the nested relational model suffers from this representational problem because it still has an associated schema. As a consequence, each nested relational table could be expressed by a XML document, but not the other way around. As a consequence, within this thesis we're going to represent nested relations as nested tables.

Data Replication vs. Object Identity

When nesting multiple relations at different levels, the need of unambiguously identify the contained objects becomes more evident. Let us observe Table 3.2, extending the previous nested relation example. As tuples describing the same entity (e.g., the product names SPAM having id 103) appear in different places within the table, their price change requires a complete scan of the whole table. Update operations within these models are quite inefficient (update of multiple instances against an update of one single instance), therefore the nested relational data model could not be used for representing hierarchical data changing through time. Moreover, since the nested relational model also suffers from semantic overloading, the description of the entity Product now also depends on the SaleOrder's quantity of ordered products. To avoid data replication problems, direct value containment should be replaced by object identifier containment. At the time of the writing, nested graphs data models are represented by the former approach (e.g., GraphML [BELP07]), while our proposed data model is going to adopt the latter solution (Chapter 5).

3.2.3 Representing graphs

Even if XML was originally designed to represent tree data structures, it could also represent graphs through specific XML schemas. **GraphML** [BELP07] and **GXL** [BLP04], which are based on the XML markup language, allow to express graphs, that could also contain other graphs (*nested graphs*) inside some other vertices and edges. We're going to analyze and compare such data models with the one we propose in Section 3.4 on page 73.

On the other hand, graphs could be also used to represent semistructured information, such as XML documents [LS99, GHMP11]. As well as XML, graphs could provide a syntactical representation (*metadata*, see Chapter 2) of the XML data: as a result, we could analyze how different authors choose to differently use the tags within some XML documents, and investigate which "structural patterns" have been used [IPPV14, BDIP$^+$13].

This fact also suggests that graphs must belong to the *semistructured data* family and that all semistructured data are able to describe both data and the meta-data level with the same description language, as well as describing query languages. We invite the reader to compare the aforementioned XSLT query language in XML with GraphQL [CM90b] pattern query language, which is expressed through graphs; other query languages represented as graphs are another graph pattern query language [FLM$^+$12] and an artificial intelligence one [GPG14].

3.3 Unstructured Data: Full Text Documents

In contrast with structured data, **unstructured data** relies on a data representation that could not be directly handled by non-domain-specific program to extract information, process it and provide results in the same format. Such kind of pieces of information include full-text documents, audio, image and video formats [SSSF09]. Within this thesis, we will particularly address the full-text documents even if some of the following illustrated techniques may also apply to the other aforementioned formats.

The need of retrieving machine readable information from full text was clear since the early days of computer science [Luh58], when IBM imagined a Business Intelligence System that was able to retrieve full text documents by using single words (*keywords*) as indices (see Subsection 3.3.1 on the facing page for more details). Anyhow, this project was too ambitious for those times, when only small full text corpora were available [Sin01]. All the research effort was then moved towards the analysis of structured documents and to develop a query language on top of it [Cod71]. During the 1990s, the first full text corpora was delivered [CHoSU93] and hence, the Information Retrieval field could finally became a meaningful research topic.

With the increase of the volume of the corpora and the rise of the World Wide Web, it became more relevant that the combination of classical Information Retrieval's keyword-indexed document based search [MRS08] with lexical similarities [KB17] was no more sufficient to achieve hight *precision* scores. As a result, at the beginning of the 2000s [Bra03] Natural Language Processing techniques such as *dependency graphs* were firstly addressed for increasing the precision of IR techniques. Computational Linguistic research developed this technique for providing a graph semantic representation of the text [VnI11]. By doing so, problems like multi-word recognition [LVJRT14], word similarity [HRJM15] and multilingual word disambiguation [NP12b] could be finally solved by using specific (multi language) knowledge bases, such as **BabelNet** [NP12a] or **WordNet** [Fel98]. Moreover, the usage of (e.g., OWL) ontologies such as **YAGO** [oRD12] and reasoning techniques on top of such graph-structured data, could even help with the increase of the recall values [WM06].

As an outcome of an NLP oriented Information Retrieval approach, it was not only possible to extract full documents satisfying some user information need, but also to extract the passages from one single document containing only the relevant information for the

Algorithm I.1 Initializing an Invertex Index for classical information retrieval queries.

```
for each $D_i \in \mathcal{D}$ do
    for each $t_j \in D_i$ do
        if $t_j \in V$ then
            $IX[t_j] \leftarrow IX[t_j] \cup \{(i,j)\}$
```

user. Such process, called **Information Extraction (IE)**, is relevant in biology, where we could extract which genes interact with each other [MZRA16] or even between different documents by using a bibliographical network [SHK+14]. This approach was also used to analyse clinical data [WKS+11], or even providing answers to open-domain questions as in the case of **IBM Watson** [oRD12] and **DeepDive** [PAKR16]. In particular, Information Extraction techniques extract graph representations of full text documents in two main phases: **Entity Extraction** and **Relation Extraction** [Sar08]. In the first phase, entities are extracted by querying the unstructured document or using some other statistical techniques. The second phase allows to extract relations among the previously extracted entities: in order to do so, grammatical relations can be preliminarily extracted from the text through dependency graphs [dMDS+14]. This last task can be more profitably used within domain specific applications, where the relationships are known beforehand [ZRC+17]. In those other use cases, universal dependencies techniques may be used instead.

Finally, graph data representation of full-texts allows to later perform either specific graph mining algorithms [SHJ+13] such as clustering [CJ10] and association rules, or basic graph metric operations, such as betweenness centrality and degree distribution [New10].

3.3.1 Query Languages

The aim of a query language is to return (either partly or as a whole) and manipulating some data. Contrary to this common sense, classical Information Retrieval [MRS08] can only return (sub)sets of documents stored in huge document collections satisfying the user's information need. This consideration also applies for more standardized IR query languages, such as Context Query Language[7], which are not able to extract relevant information from the given document.

▸ **Definition 11** (Classical Information Retrieval). *The ground truth is composed of a document collection $\mathcal{D} = \{D_1, \dots, D_n\}$. Each document $D_i \in \mathcal{D}$ is indexed using a set V of relevant terms ($t \in V$), called* ***vocabulary****. Each document $D_i \in \mathcal{D}$ is defined as a list of consecutive* ***terms*** *$D_i = \{t_1, \dots t_{m_i}\}$. Each document is indexed using the vocabulary terms through* ***inverted indices*** *IX: such index associates each term $t \in V$ to a set of pairs (i,j) defining that the term t occurs in document d_i as the j-th term. Such indices could be initialized as showed in Algorithm I.1.* ◂

Classical Information Retrieval does not provide a formalization of how to translate an user's full text query into a formal language. To make matters worse, some approximated frequency-based approaches such as TF-IDF are ill defined so that, in the worse case scenario, negative frequencies could be obtained[8]. For this reason, only exact information

[7] http://www.loc.gov/standards/sru/cql

[8] *"The counter-intuitive negative weights referred to in section 1.3 would normally arise only in the case of a term which occurred in a very large portion of the collection. As this is a very rare occurrence in most collections, this has not been seen as a problem."* [RW97]

retrieval approaches will be considered within this thesis. In particular, the "Boolean IR Query" language could be formalized as follows:

▸ **Definition 12** (Boolean IR Query). *A full text query ftq is defined as follows:*

1. *A string "k", where k is a string (keyword) containing no empty characters, is a ftq.*
2. *A string "$k_1 \quad k_2 \quad \dots \quad k_n$", containing n space separated keywords is a ftq.*
3. *"ftq_1 **AND** ftq_2" is a ftq.*
4. *"ftq_1 **OR** ftq_2" is a ftq.*
5. *"(ftq)" is a ftq.*

∘. *Nothing else is a ftq.*

Given an inverted index IX, the interpretation $[\![ftq]\!]^{IX}$ of a ftq returns the set of documents satisfying ftq:

1. $[\![k]\!]^{IX} = \{\, D_i \mid \exists j.(i,j) \in IX[k]\,\}$
2. $[\![k_1 \quad \dots \quad k_n]\!]^{IX} = \left\{\, D_i \;\middle|\; \exists j. \bigwedge_{h=1}^{n} (i, j+h-1) \in IX[k_h]\,\right\}$
3. $[\![ftq_1 \textbf{ AND } ftq_2]\!]^{IX} = [\![ftq_1]\!]^{IX} \cap [\![ftq_2]\!]^{IX}$.
4. $[\![ftq_1 \textbf{ OR } ftq_2]\!]^{IX} = [\![ftq_1]\!]^{IX} \cup [\![ftq_2]\!]^{IX}$.
5. $[\![(ftq)]\!]^{IX} = [\![ftq]\!]^{IX}$.

◂ ◂

We now want to provide an example to show that the aforementioned semantics for a Boolean IR query is the usual intended meaning for a full-text query.

▸ **Example 16.** *The following table represent an example of document corpus that could be used for information retrieval:*

Document Id	*Content*
D_1	*The quick brown fox jumps over the lazy dog*
D_2	*Jack be nimble, Jack be quick. And Jack jump over the candle stick*
D_3	*And it seems to me you lived your life like a candle in the wind.*

If we want to perform the query "quick", then we could easily see that it evaluates to $\{D_1, D_2\}$, while "candle" evaluates to $\{D_2, D_3\}$ because the inverted index IX from the document collection is accessed and the stored values are simply returned.

*At this step, the interpretation of the query "quick" **AND** "candle" we have that such query evaluates to $\{D_2\}$ because in that document both terms appears. Last, the query "quick candle" returns no document because in no document those two words appear consecutively.*

The previously defined query language is so simple that we cannot extract multiterms and avoid some intermediate characters. For these reasons, cascading grammar rules have been considered for term extraction techniques, jointly with dictionary and regex matching. Nevertheless, such approach have been supplanted by relational algebra techniques [RRK+08], providing a "semantic" for such grammar rules that could boost their performance by using relational algebra rewriting rules. This implies that we must provide a structured representation for the unstructured document.

▸ **Definition 13** (Span). *Given a document D represented as a collection of characters $c_1 \dots c_n$, a **span** [RRK+08] is an interval represented as a pair (i,j) identifying a term $c_i \dots c_j$ within the document. Each span could be represented as a tuple $t = (i,j)$ of a relation $r(R)$ having schema $R = (\texttt{begin}, \texttt{end})$.* ◂

After obtaining such spans through either regex-es or dictionary extraction techniques over a full text document (*span extraction operators*), we can aggregate such spans with given rewriting rules by composing the previously extracted term (*span aggregation operators*) Even though such last operators could be still expressed through a composition of the standard relational operators plus a *while-loop* operator as the one described in [CLNP06], they were implemented as distinct operators for efficiency reasons.

To the best of our knowledge, the only other algebra over full text documents that has been defined is the one for retrieving and querying full text information within semistructured documents [BM08] but again, such algebra is not able to recombine textual contents but only to filter and score them as the other IR query languages.

3.4 Graph (Data) Models

Graph data models directly overcome the semantic overloading problem by distinguishing entities and relationships, respectively represented as vertices and edges. In particular, graphs G are represented as pairs (V, E), where V is the vertices' set and $E \subseteq V \times V$ is the edges' set. This popular theoretical data structure has been variously extended to support data representations. Such solutions are going to be discussed in the following paragraphs.

Property Graph

Property graphs represent multigraphs (that are graphs where multiple edges among two distinct nodes are allowed) where both vertices and edges are multi-labelled tuples. This model does not allow the storage of aggregated values, both because values in U cannot contain either vertices or edges, nor U is made to contain collection of values. This data model is implemented in almost all recent Graph DBMSs, such as **Neo4J** [RWE13] or **Titan**. We discussed this data model in the previous sections, where it was compared to other non-graph data models. Given that we have already observed that this data model is complementary to the previous "nested" ones, we now want to check whether other graph data model extensions support nested graph contents.

RDF Model

This other graph data model is used in the semantic web and in the ontology field to describe Linked Data [FPG15, HP15]. Consequently, modern reasoners such as **Jena** [CDD+04] or **Pellet** [SPG+07] assume such data structure as the default graph data model.

▸ **Definition 14** (RDF (Graph Data) Model)**.** *An **RDF (Graph data) model** [GHMP11] is defined as a set of triples (s, p, o), where s is called "subject", p is the "predicate" and o is the "object". Such triple describes an edge with label p linking the source vertex s to the destination vertex o. Such predicate can even appear as a source vertex whenever additional information is provided [DSP+14]. Each vertex is either identified by a unique URI identifier or by a blank node b_i. Each predicate is only described by an URI identifier.* ◂ ◂

Even if this data model provides unique resource identifiers as a common basis, it does not allow to store some attribute-value information inside each node. Consequently, each entity's attribute is mapped as an edge linking the resource to its property. [DSP+14] shows that property graphs can be entirely mapped into RDF triplestore systems as follows:

▸ **Definition 15** (Property Graph over Triplestore). *Given a property graph $G = (V, E, A, U, \ell, \kappa, \lambda)$, each vertex $v_i \in V$ induces a set of triples (v_i, α, β) for each $\alpha \in A$ such that $\kappa(v_i, \alpha) = \beta$ having $\beta \neq$ NULL. Each edge $e_j \in E$ induces a set of triples (s, e_j, d) such that $\lambda(e_j) = (s, d)$ and another set of triples (e_j, α', β') for each $\alpha' \in A$ such that $\kappa(e_j, \alpha') = \beta'$ having $\beta' \neq$ NULL.* ◂

The inverse morphism is not always possible because RDF properties can be even used as either source or targets for other properties, while edges within the property graph model can be only used to link other vertices. Last, this RDF also support *named graphs*, through which graphs are associated to a resource identifier; in particular, each property graph may be stored as a distinct named graph. Even though this model allows to use such named graphs as subjects, they cannot be used as neither objects or properties. Therefore, such model does not overcome property graphs' limits.

Extended Property Graph Model (EPGM)

The need of representing both graphs and graph collections for handling pattern matching queries presented for GraphQL [He07] brought to the definition of data models where both representations are provided. Therefore, the property graph data model requires to be extended because property graphs do not natively support graph collections. For this reason both GRADE [GRS+16, GRS+15] and EPGM [JPT+16, JPR17] were introduced: within this thesis we're going to describe only the latter one due to its practical implementation in Gradoop. This data model can be re-defined[9] as follows:

▸ **Definition 16** (EPGM Database). *An EPGM database $DB = (V, E, L, K, T, A, \lambda, \nu, \varepsilon, \kappa)$ consists of vertex set $V \subseteq \mathbb{N}$, edge set $E \subseteq \mathbb{N}$ and a set of logical graphs $L \subseteq \mathbb{N}$ such that those identifiers' sets are pairwise disjoint. An edge $e \in E$ is mapped to its source and target vertices through the λ function, e.g. $\lambda(e) = (s, t)$, where $\lambda \colon E \to V^2$. To each* ***logical graph*** *$g \in L$ is associated a set of vertices and edges through the ν and ε functions, such that $\nu(g) \subseteq V$ and $\varepsilon(g) \subseteq E$, and each extracted edge connects edges within $\nu(g)$ $(\forall e \in \varepsilon(g).\lambda(e) \in \nu(g)^2)$. Vertex, edge and graph properties are defined by key set K, value set A and mapping $\kappa : \mathbb{N} \times K \to A$. Labels T are expressed as a value A associated to a key $\tau \in K$.* ◂

Even though this data model was used to express graph aggregations, neither logical graphs nor vertices could be directly used to describe the structural content of summarized graphs. Logical graphs were not used for linking aggregated informations because EPGM does not allow any primitive object acting as a relation between either logical graphs or vertices. Moreover, neither vertices nor edges can be used for structural aggregation because, as for the property graph model, A values can represent neither object identifiers nor collections. As a consequence, ancillary "super-vertices" and "super-edges" were used to store the result of the aggregation, consisting respectively of a collection of vertices and edges. Therefore, even this definition fails at representing nested graphs, where it is required that each component contains a whole graph as represented in the former Figure 2.11 on page 33.

[9] From now on, the data models are re-defined using the same notation and terminology, in order to remark the similarities with the previous models. In particular, ν and ε are introduced as containment functions for vertices and edges, and λ is used to associate to each edge its source and destination vertices. Later on, we're only going to use ϕ for both vertex and edge containments.

Statechart and Hypernode models

Statecharts [Har87] represent one of the first applications of nested graphs for complex systems modelling. This choice allowed the representation of multiple abstraction levels at the same time: each node represents a state or "configuration" of the system, and each edge represents a transaction between two different states on a given event. In order to represent different nesting levels, each node may contain other states and edges. As a consequence, there is no distinction between (simple) states and states containing other states. Given that this model was not designed for data representations, vertices and edges are labelled but cannot contain any property-value association. Therefore, we can say that each vertex can represent both a simple vertex or a logical graph as within the EPGM Database.

▸ **Definition 17** (Harel's Statechart). *An **Harel's statechart** is a labelled multi-graph $(V, E, \ell, \lambda, \nu, \varepsilon)$, where $\ell\colon (V \cup E) \to \Sigma$ is the vertex and edge labelling function and $\ell\colon E \to V^2$ is the function associating to each vertex its source and destination. $\nu\colon V \to \mathcal{P}(V)$ and $\varepsilon\colon V \to \mathcal{P}(E)$ are the vertex and edge containment function defining which vertices and edges are contained in V; in particular, each edge e contained by a state v must link at least one vertex[10] within v.*

In particular, we say that a vertex v is contained at the n-th nesting level of a vertex u iff. $v \in \nu^n(u)$: since each state represents a different abstraction level, each vertex cannot contain itself in any abstraction level ($\forall n \geq 1. v \notin \nu(v)$). Last, since one internal state shall be contained only once because each state has an unique representation, the vertices' vertex content shall be mutually disjoint ($\forall u, v \in V. \nu(u) \cap \nu(v) = \varnothing$). ◂

We can say that Figure 2.11 provide an example for statecharts. This model allows both **external edges** and **internal edges**[11]: an edge e will be called *external* if its source (or target) is contained by the target (or source) but neither of them contains e[12]; the edge will be called *internal* when the containing vertex (either its source or target) also contains the edge[13]. Besides of state representation purposes, this model has also been used for both modelling the evolution of *pathophysiological* states and to describe the clinical treatments to which the patient must undergo. This model also allowed to subdivide each treatment into smaller and consequential procedures [KW82].

Statecharts were also adopted as a basis for the subsequent **hypernode** data model [PL94]. Unlike statecharts, hypernodes allow neither edge labelling nor external and internal edges. As previously stated for statecharts, even this model does not allow to fully represent a property graph, since the attribute-value association must be necessarily expressed as a relation between two different vertices. The fact that the vertex containments cannot overlap make such nested model affected from the same *data replication* representation problem described for semistructured and nested data (Section 3.2.2 on page 68). A first extension of the hypernode model towards data representation is represented by CoGITaNT [GS98], where any type of edge (thus including internal and external ones) are included and data is firstly contained inside a node. Nested graphs can be serialized in both GraphML [BELP07] and GXL [HSSW06] formats.

[10] Formally, $\forall e \in \varepsilon(v). \exists u \in \nu(v). \exists u' \in V. \lambda(e) = (u, u') \vee \lambda(e) = (u', u)$

[11] Since each edge provides a state transition, the *external edge* was originally called called *external transition*. Similar consideration follows for *internal edge*.

[12] Formally, $\exists n \geq 1. \lambda(e) = (s, t) \wedge ((s \in \nu^n(t) \wedge e \notin \varepsilon(t)) \vee (t \in \nu^n(s) \wedge e \notin \varepsilon(s)))$

[13] Formally, $\exists n \geq 1. \lambda(e) = (s, t) \wedge ((s \in \nu^n(t) \wedge e \in \varepsilon(t)) \vee (t \in \nu^n(s) \wedge e \in \varepsilon(s)))$

Graph Data models for roll-up and drill-down operations

As outlined in the Introduction at page 12, graph representation have recently became a favourite representation for multidimensional data. Current literature uses two different approaches for extending graph databases to support nesting operations: some try to overcome graph data structure limitations by extending their query languages, while others try to extend the data structures used for both input and intermediate computations. Among the first type of approaches, [EV12b] proposes the definition of a RDF vocabulary over which the OLAP cube can be defined. On top of this "structured" RDF graph, an algorithm generates the SPARQL query that will allow to perform either the roll-up or the drill-down operation. This implies that each possible computation over the data view has to be always recomputed on top of the raw data as in ROLAP systems, thus thwarting the benefits of updating the intermediate query result. On the other hand, the last type of approaches has been recently widely investigated and seems to be more promising with regard to optimization techniques. In these approaches [THP08, CYZ+08, QZY+11], graph data structures are associated with external graph indices and, thus, allow to connect one graph to a broader one with respect to the roll-up query. As a consequence, these solutions do not allow to freely expand any aggregate components at the same time but can only backtrack the aggregation to a previous known state.

As it will be showed in Chapter 5, in order to meet such goals the nesting indices are going to be directly embedded within the definition of the nested (graph) data model, thus allowing to extend all the aforementioned approaches.

3.5 Classifying Graph Query Languages

Contrariwise to current graph query languages' surveys [AAB+17], we're going to classify them not only from their expressiveness and ability to perform several forms of traversal and matching queries but also by their ability to generate new graph data. Therefore, graph Query Languages can be categorised in three main classes:

1. The first class tries to find a possible match for a specific traversal or for extracting all the subgraphs that match a given pattern (Section 3.5.1 on the next page). Such graph queries do not necessarily manipulate the graph data structure, but they traverse it using data properties which select the pieces of informations that must be returned after the visiting process.
2. Graph grammars are used to select parts of a same graph and create new vertices and edges by using rewriting rules (Section 3.5.2 on page 79). These languages provide the possibility of adding and removing new vertices and edgesh which do not necessarily depend on previous data.
3. Graph algebras extend the previous operations with set and simil-relational algebra operators for graphs (Section 3.5.3 on page 80). These languages permit n-ary operators and the modification (projection, extension) of already-existing objects. Some graph algebras also provide operators accepting as inputs (or returning) graph collections.
4. The last class of languages are "proper" graph query languages, that include within their expressive power all the aforementioned class of operations (Section 3.5.4 on page 82).

Graph Language	Query	Result
Wenfei Fan et al. [FLM$^+$12]	Graph + REGEX	1 single transformed matched graph
GraphLOG [CM90a, CM90b]	Graph + REGEX	1 single transformed matched graph
Isomorphism	Graph	Morphisms defining a collection of matched subgraphs
NautiLOD [FPG15]	REGEX	Matched graph (or vertex) collections
Description Logic [BCC$^+$16, BCM$^+$10]	Description Logic	Vertices from which the graph can be matched
NLR$^+$ [BFL13]	REGEX + registers	Register-based morphisms over satisfiability
Gremlin [Rod15]	Gremlin	Bag of values $V \cup E$ and side effects
HyperLog [PH01]	Clauses	Hypernode
XPath [BBC$^+$15]	REGEX	Subtrees reached after traversing the expression

Table 3.3 An example of graph traversal and graph pattern matching query languages: as we can see, there are huge differences between those graph query languages, both on the query representation and on the provided result and on its semantics.

3.5.1 Graph Traversal and Pattern Matching Languages

In present literature there are two distinct types of languages allowing the subgraphs "extraction" from a single graph operand: the first approach is to write an expression in a given language $\mathcal{L}$ such that its interpretation involves a visit of the graph, while the second approach is to express such query with another graph (eventually enriched with other path expressions), thus directly providing the data structure to be searched within the graph. In both cases, there are solutions allowing to perform the graph visit via "tractable" algorithms, such that the graph visit happens at most in a polynomial time with respect to the graph data size [FLM$^+$12, FPG15, BFL13]. Consequently, such solutions do not necessarily involve to run a subgraph isomorphism problem, except when expressly requested by specific semantics [AAB$^+$17, JKA$^+$17].

Table 3.3 shows that there is no strict classification under which we can label such graph languages, even though we can observe that they all rely on a same mechanism: each query has to be interpreted by a specific semantics transforming such query into an intermediate language expression (e.g. a generic program) over which the graph data input can be provided as an input and returned as an output. Consequently, we can just distinguish these query languages by the result they provide, either a single graph or a collection of graphs. For both languages we introduce a new denomination, and call the former "graph selection languages" and "graphs extraction languages" the latter.

3.5.1.1 Graphs Extraction Languages

Since the graph data structure does not allow an uniform representation for both graphs and graph collections, we represent graph extraction languages as an operator from graph to graph collections (which are here expressed via a collection of "morphisms"). Such morphisms can be expressed as single tuples within one relational table [AAB$^+$17].

▸ **Definition 18** (Graphs Extraction Languages). *Given a* ***graphs extraction language*** *$\mathcal{L}$ and a query $Q \in \mathcal{L}$ expressed as a graph or a regex (or both), the interpretation $m_Q(G)$ of Q over an input graph G returns a set of functions f_i called* ***morphisms*** *mapping each component of P into a list of* *either vertices or edges in G in $\wp(V_G \cup E_G)$. In particular:*

$$m_Q(G) = \{f_i : Q \to \wp(V_G \cup E_G)\}_{i \leq n}$$

◂

Gremlin

Gremlin is a Turing Complete graph traversal query language [Rod15]: this is not a desired feature for query languages since they must usually guarantee that each query evaluation must always converge and that it must always return an answer in a "reasonable" amount of time. Another problem with this query language is based on its path navigation semantics [TPAV17]: while all the other graph traversal languages return the desired subgraph, Gremlin returns a bag of values (e.g. vertices, values, edges). This peculiarity does not allow the user to take advantage of partial query evaluations and to combine them in a final result. This feature is also shared by other other path navigation algebras, such as the one for Cypher[14] [HG16, MSV17], which algebra only considers the graph traversal aspects of the language.

3.5.1.2 Graph Selection Languages

Instead of returning multiple graphs matching one single query Q, other languages prefer to return the maximal subgraph containing all the graphs that match Q.

▸ **Definition 19** (Graph Selection Language). *Given a **graph selection language** $\mathcal{L}$ and a query t for such language expressed as a graph (or both), the interpretation $[[t]]^{\mathcal{L}}$ of t over an input graph G returns, it it exists, an (homo)morphism $\varepsilon : \gamma \to t$ such that $\gamma \subseteq G$ is the least upper bound[15] of all the subgraphs γ' of G satisfying t:*

$$[[t]]^{\mathcal{L}}(G) = \left\{ \varepsilon : \gamma \to t \;\middle|\; \gamma = \sup(\{\gamma' \subseteq G | t(\gamma')\}), \varepsilon(\gamma) = t \right\}$$

Since this operation acts as a selection function for the graph, the returned graph can be expressed through the following selection operator:

$$\sigma_t(G) = \begin{cases} (V \cap \mathrm{dom}(\varepsilon_V), E \cap \mathrm{dom}(\varepsilon_E)) & \varepsilon \in [[t]]^{\mathcal{L}}(G) \\ (\varnothing, \varnothing) & oth. \end{cases}$$

◂

As a consequence, this selection operator appears to be more general than the graph selections provided in graph literature [JPT+16], where there are only predicates over vertices and edges, that are included by the aforementioned graph traversal operator, from now on called "selection". I now provide some examples of graph selection languages in the following subsections.

[14] Please note that the algebra [TPAV17] for Gremlin and the other one [HG16, MSV17] for Cypher are substantially the same, with some minor changes.

[15] An *upper bound* for X in poset $(S, \leq)$ is an element $M \in S$ such that $\forall x \in X. x \leq M$. T is also a **least upper bound** for X in S, denoted by $\sup(X)$, if $\forall x \in X. x \leq T \wedge \forall d \in X. (\forall x \in X. x \leq d) \wedge T \leq d$.

Description	Syntax $(\cdot)$	Semantics $(\llbracket\cdot\rrbracket(u))$
Entry point	$\cdot p$	$\bigcup\{\Gamma\in\llbracket p\rrbracket(u)\mid T_\Gamma\neq\varnothing\}$
Edge traverse	ℓ	$\bigcup_{u\xrightarrow{\ell}v}(\{u,v\},\{u\xrightarrow{\ell}v\},u,\{v\})$
Inverse edge traverse	$\ell\hat{}$	$\bigcup_{v\xrightarrow{\ell}u}(\{u,v\},\{v\xrightarrow{\ell}u\},u,\{v\})$
Any edge	$\langle\_\rangle$	$\bigcup_{\ell\in\mathcal{U}}\bigcup_{v\xrightarrow{\ell}u}\llbracket p\rrbracket(u)$
Vertex predicate θ	$p[\theta]$	$\bigcup\Big\{(V,E,s,\{v\in T\mid\theta(v)\})\;\Big\vert\;(V,E,s,T)\in\llbracket path\rrbracket(u)\Big\}$
Traversing	p/q	$\llbracket p\rrbracket(u)\circ\Big(\bigcup_{v\in T_\Gamma,\Gamma\in\llbracket p\rrbracket(u)}\llbracket q\rrbracket(v)\Big)$
Disjunctive Path	$(p\vert q)$	$\llbracket p\rrbracket(u)\cup\llbracket q\rrbracket(u)$
Kleene Star	$(p)^*$	$(\{u\},\varnothing,u,\{u\})\cup(\bigcup_{i=1}^{\infty}\llbracket\underbrace{p/\dots/p}_{i}\rrbracket(u))$

Table 3.4 SUCCESSFUL semantics S associated to each query $\cdot p$ expressed in the NautiLOD syntax. Side effect *action* evaluations were removed. u denotes the initial source vertex from which the graph traversal query is started.

NautiLOD

The **NautiLOD** [FPG15] query language was conceived for performing path queries (defined through path expressions with REGEX-es) over "RDF graphs". Notwithstanding the usage of recursion operators via Kleene Star, the same paper shows that queries can be evaluated in polynomial time. While other graph pattern matching query languages provide as an outcome of their evaluation a blob of both graph vertices and edges (e.g. Gremlin), its interpretation can be mapped into a combination of algebraic graph operators, through which a graph collection is provided in return.

NautiLOD uses MULTIPOINTED GRAPHS to express the result of the query evaluation: each MPG= (V,E,s,T) is a RDF graph extended with a source vertex s and a target set T : s represents the vertex from which the graph traversal is started, and T represents one of the possible ending nodes. The interpretation of such graph traversal semantics is expressed using three basic MPG operators, that can be extended to MPG collections. In particular, $mpg_1\circ mpg_2$ expresses a path concatenation, where the two graphs are united only if s_2 appears in T_1; the union[16] $mpg_1\sqcup mpg_2$ either performs the union of two non-empty MPGs sharing the same source vertex, or returns one of the two empty graphs; $\bigcup$ denotes[17] the union of two MPG sets. Table 3.4 provides an example of how such operators provide the query interpretation using the graph and graph collection operators. We're going to discuss this query language's interpretation over our proposed data model at page 179, where we're going to use it for traversing nested graphs.

3.5.2 Graph Grammars

Graph grammars are similar to grammars generating programming languages: the latter grammars are defined through a collection of "rewriting rules" $H\to T$, called *productions*, where H is a non-terminal symbol while T is a collection of both terminal and non-terminal symbols. In particular, $H\to T$ means that each occurrence of H produced while expanding

[16] Originally denoted as $\cup$.
[17] Originally denoted as $\oplus$.

the rules must be replaced by T. Similarly, graph grammars are expressed through "rewriting rules" allowing to re-write a subgraph into one subgraph, while keeping the former connections valid. In particular, this thesis will consider single-pushout graph grammars formalized over traditional graphs as follows:

▸ **Definition 20** (Graph Grammar). *A single-pushout graph grammar $\mathcal{G}$ [Ren03, Löw93] is a collection of productions $G_H \overset{\varphi}{\to} G_T$, where G_H is the pattern graph and G_T is the rewriting graph, and φ is the identity function mapping all the vertices and edges in G_H that are preserved in G_T.*

For each production rule $G_H \overset{\varphi}{\to} G_T$, given a morphism φ mapping the subgraph $\gamma \subseteq G$ of a given graph $G = (V, E)$ into the pattern graph G_H, the rewriting of G is defined as the graph $(V_\tau, E_\tau|_{V_\tau})$, where the vertex set V_τ and E_τ are defined as follows:

$$V_\tau = (V_{G_T} \backslash \mathrm{cod}(\varphi)) \cup \varphi(V_G)$$

$$E_\tau = (E_{G_T} \backslash \mathrm{cod}(\varphi)) \cup \varphi(E_G)$$

while $E_\tau|_{V_\tau}$ is a restriction of all the edges in E_τ wth V_τ as defined in [Ren03] where all the source and destination edges appear in V_τ. ◂

On the other hand, these languages do not guarantee to be invariant to the order of the application of the grammar rules $\mathcal{G}$ over the graph input, and hence different production rules application strategies can be carried out to provide the desired result. This problem still appears in current graph query languages implicitly adopting such techniques (e.g. Cypher), which internally apply the graph grammar's rewriting rules while generating new graphs. Moreover, such formalizations only applies when G_H actually matches a subgraph of the input graph operand.

Last, there are a few problems in applying such technique to current graph query languages: the first aspect is that the graph G_H is often used as a pattern matching query, and hence it often does not represent the actual data stored in a graph G, but some general features that have to be present. Moreover, since such graph pattern matching techniques can return more than one possible match through one or more morphisms, such techniques fails at rewriting graphs in G_T using the different morphisms. We're going to later on generalize such concept in Section 6.3.4 after studying the other query languages and, above all, (proper) graph query languages, where these graph rewriting concepts are used even if they haven't been formalized yet.

3.5.3 Graph Algebras

Within the field of database query languages, a (relational) algebra is a data structure (e.g., finitary relations in the case of Codd's *relational algebra*) that is closed under certain operators. The aim of such languages is to provide a semantics and optimizations to declarative programming languages (e.g. SQL for the *relational algebra* [CG85]) such that it can be used within the databases' query plans to obtain, through their rewriting rules (i.e. *equational reasoning*), the most efficient query providing the equivalent result, independently from the data actually stored within the database. Moreover, algebras express graph transformation that are either impossible or hard to provide through graph rewriting, such as the basic graph set operators (union, intersection, difference over the vertices' and edges' set) and binary graph operators (join).

Currently there are two types of algebras that have been developed for graphs: path algebras and (proper) graph algebras. While path algebras have been developed to provide a formal semantics to path traversal and graph pattern matching queries, graph algebras have been designed either to change the structure of property graphs through unary operators, or to combine then through binary ones (e.g., the graph join and graph nesting that will be respectively introduced in Chapter 4 on page 89 and 7 on page 195). Among all the (graph) algebras, only path algebras have been studied for both expressing equivalence rules (for SPARQL, [PAG09]) for providing path evaluations' optimizations (for Cypher, [HG16]) and allowing incremental updates (for SPARQL, [Shm11]). On the other hand, due to the fact that such algebras were only designed for path manipulation and not for returning graphs, they cannot be directly used to return graphs.

Property graphs (and their extensions) also fail both to be closed under their proposed (proper) graph algebras, and to provide some interesting rewriting rules. The reason for this is twofold: firstly, the need to express graph collections and graphs for some graph data analysis tasks [JPT+16] and the impossibility of expressing them both within the same representation under the property graph model requires the algebras to design four distinct class of operators: graph to graph operators, graph to graph collection operators [JKA+17], graph collection to graph collection operators [He07], and graph collections to graph operators [JPT+16]. Therefore, no simple rewriting rules can be modelled on top of such algebras (GRAD [GRS+16, GRS+15] and GrALa [JPT+16]): such problem was originated by their ancestor GraphQL[18] [He07]. Moreover, those algebras were designed more to be useful APIs to the programmers than actual tools for query optimization: this intuition is supported by their implementation, focusing more on the algorithmic enhancement of the single operator and not on how such operators interact [JPR17]. Moreover, the graph collection to graph collection operators do not implement all the possible collection operators as outlined in [MM06]: as an example, proper join operations are missing.

Last, some of the path algebras do not distinguish data from the operator themselves, and hence data became actually 0-ary operators [HG16, MSV17, TPAV17], that have no sense with respect to standard algebras, where the data itself is not expressed as an operator.

GraphQL, GrALa and GRAD

GraphQL [He07] is yet another graph query language with an SPARQL-like syntax, mainly conceived for pattern extraction from the data, called *graph motifs*, and their construction. The language allows graphs naming similarly to SPARQL *named graphs*. The most interesting scientific contribution of He [He07] is the first attempt in defining a graph algebra for collection of graphs.

This approach has been finally specialized for single graphs in the **GRAD** algebra [GRS+16, GRS+15]. In this latter definition the *cartesian product* and *join* operations are still defined over graph collections and are still not specialized for the single graphs. Consequently, in both languages the cartesian product over two graph collections produces a graph containing two (possibly) disjoint graph components. The graph join over the two collections only merges the matched vertices and no considerations are made on the graphs' edges structure. In the end, GRAD propose an alternative graph data model that could be expressed as a specific implementation of the Property Graph model.

On the other hand, graph joins are completely missing on the other development of the GraphQL algebra, that is **GrALa** over EPGM [JPT+16]: its graph pattern matching through

[18] Not to be confused with Facebook's GraphQL query language [HP17].

graph motifs inspired the recent pattern matching operator [JKA+17], acting both as a graph traversal and as a proper pattern matching query interpretation on top of Cypher's query syntax. Last, the dichotomy between graph and graph collection inspired the authors to extend the basic property graph model to support graph collections within a same graph database. The definition of graph operators for such algebra is still under development.

3.5.4 (Proper) Graph Query Languages

A graph query language is "proper" when its expressive power includes all the aforementioned query languages, and possibly expressing the graph algebraic operators. In particular, such languages are able to express graph grammars' rewriting rules for both updating the data within the given graph database, and creating new elements:

- in Cypher, the following keywords are used: `SET` for setting new values within vertices and edges, `MERGE` for merging set of attributes within a single node or edges, `REMOVE` for removing labels and properties from vertices and edges, and `CREATE` for the creating of new nodes, edges and paths [Inc14].
- in SPARQL, `INSERT` and `DELETE` clauses allow to create and remove RDF triplets [SGP08].
- in HyperLog [PH01] uses a Prolog-friendly syntax with negations, which are used to remove the matched vertices or edges. Such query language also allows the creation of new hypernodes.

Moreover, they can express graph traversal queries [KRRV15], set operations and pattern matching ones [JKA+17]. Even if graph grammars seems to arbitrarily extend the expressive power of such query language by allowing to express some new graph operators, in some cases a direct algebraic implementation proves to be more efficient than the pattern matching (or graph traversing) plus graph transformation mechanism. This intuition was proved for the graph join operator that is going to be presented in Chapter 4 on page 89, where a straightforward implementation of such operator proved to be more efficient than the query matching and rewriting.

Last, even though these languages can be closed under either property graphs or RDF, graphs must not be considered as their main output result, since specific keywords like `RETURN` for Cypher and `CONSTRUCT` for SPARQL must be used to force the query result to return graphs. Given also the fact that such languages have not been formalized from the graph returning point of view, such languages prove to be quite slow in producing new graph outputs.

The following paragraphs will provide a round up of the features of such proper graph query languages.

SPARQL

At the time of writing, the most studied graph query language both in terms of semantics and expressive power is **SPARQL**, as it is the most time-worn language among those that are both well-known and implemented. Some studies on the expressive power of SPARQL [AG08, PAG09] showed that it allows to write very costly queries that can be computed more efficiently whether only a specific class of (equivalent) queries is allowed. As a result, the design flaws of a query language relapse on the computational cost of the allowed queries. These problems could be avoided from the very beginning whether the formal study had preceded the practical implementation and definition of the language. However, such limitations do not preclude some interesting properties: the algebraic language used

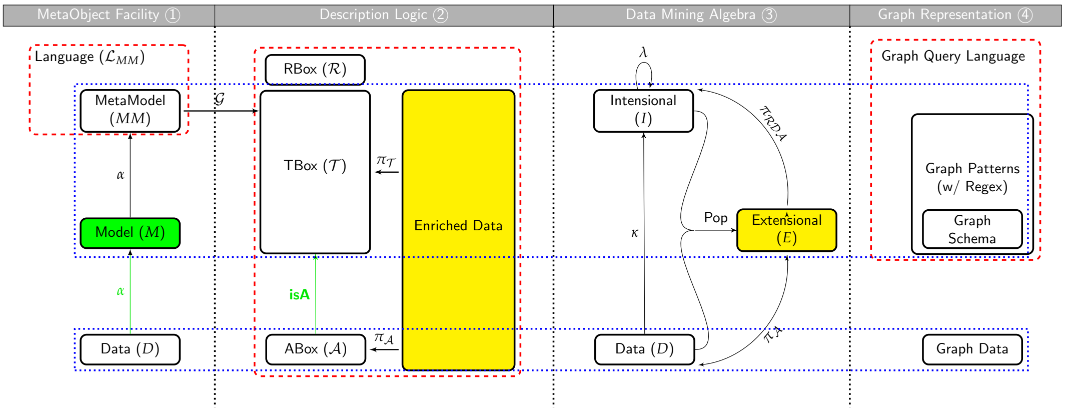


Figure 3.6 A coarse view of all the query languages that have been analysed so far. Red dashed lines represent the level at which the query languages lie with respect to the representation of the data to be queried. Blue dotted lines represent the level at which either data (below) or its properties (above) are represented.

to formally represent SPARQL performs queries' incremental evaluations [Shm11], and hence allows to boost the querying process while data undergoes updates (both incremental and decremental). Anyway, a lot of research has been carried out [PAG09] and efficient query plans have been implemented [HAR11], even when multiple graphs are took in input. These results involve the interpretation and the execution of "optional joins" paths [Atr15], thus allowing to check whether the graph conjunctive join conditions are not met for the outgoing edges. While SPARQL was originally designed to return tabular results, later extensions (SPARQL 1.1) tried to overcome to such problem with the `CONSTRUCT` clause, that returns a new graph (see the query at page 62). While the clauses represented within the `WHERE` statement are mapped to an optimisable intermediate algebra, such considerations do not apply for the `CONSTRUCT` statement. However, `CONSTRUCT` is required for produce a graph as a final outcome of our graph join query. Last but not least, the usage of so-called *named graphs* allows the selection of over two distinct RDF graphs.

Cypher

Cypher [Neo13, RWE13, Inc14] is yet another SQL-like graph query language for property graphs. No formal semantics for this language were defined from the beginning as in GraphQL, but nervelessly some theoretic results have been carried out for a subset of **Cypher** path queries [HG16] by using an algebra adopting a path implementation over the relational data model. Similarly to SPARQL, such algebra does not involve graph creation processes within the `CREATE` clause. Such solution is also reflected by its query evaluation plan, which provides a relational output as a preferred result; as a result, the process of create new vertices and edges is not optimized. Such language allows to update a property graph and to produce a new graph as a result.

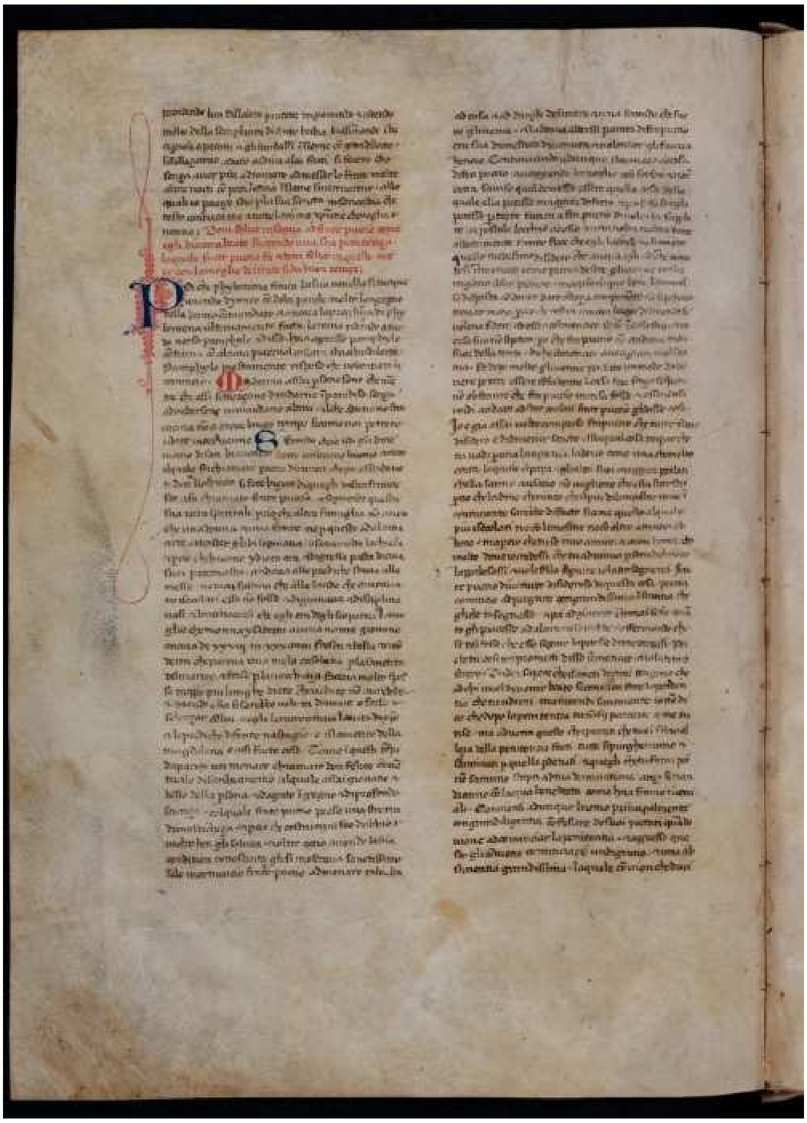
3.6 Conclusions

Structured and semistructured models point out that a schema can be externally represented, and that (semi)structured data can be represented without the constraints provided by a specific schema, thus allowing a more flexible approach to data integration between different representations. Moreover, the analysis of unstructured data revealed that their translation into a representation of choice depends on a specific scenario, similarly to the adoption of a specific schema to semistructured data. On the other hand, all the non-graph models fail to represent explicit relations between different data and are all affected by semantic overloading: this last problem is solved by the property graph model. Additionally, these data model representations failed to provide structural aggregations: the only model achieving this final result was the stream data model: any summarized representation is directly associated to the finer values, and each value may be also contained in multiple possible coarser representations. As a consequence, a proper model extending both the semistructured data representation and the graph model is required: Chapter 5 on page 119 is going to propose GSMs where associations between attribute and value for each tag, tuples and objects are going to be generalized into a `MultiMap` of object references instead of a single `Map`; GSMs will also enable structured aggregations.

Finally, we completed the analysis of several types of query languages: those are roughly subsumed by Figure 3.6. ① At the very beginning of this thesis, we were trying to find a valid representation of semi-structured data, and recognized that the data model itself shall contain a subset MM of the whole query language $\mathcal{L}_{MM}$. Still, at this abstraction level it was not clear which query language should have been used. Then, ② we analysed the Description Logic framework (Definition 4 on page 48), which is currently used for data integration, and we observed that, even though this language can be used to model data alongside its properties, it fails to reproduce data transformation operations. Then, we moved to an ③ relational algebra extension for data mining (Section 3.1.1.1 on page 58), where it was also possible to transform the tuples' representations and to perform aggregations. Nevertheless, this query language showed the impossibility one singe relational representation for both data and intensional data properties, thus requiring the explicit definition of extra-world operations. Then, we moved to analyse ④ graph query languages: we introduced graph grammars, introducing the pattern matching and rewriting process that happens in proper graph query languages; such operations both represent most of the possible operations over graph data structures and perform schema alignments and rewritings. We also observed that their instance-specific head rules must be substituted with more general graph pattern matching or traversing queries, thus allowing to nest vertex and edge properties within graph data structures. This observation also suggests that graph data and graph schemas/patterns may be represented using the same data model, thus suggesting that a generalized graph data model should be also able to embed metamodel information. Consequently, we expect that the GSQL query language (Chapter 6) on top of GSM will also express all the previous types of queries, as well as embed the GSM's metamodel within its operators.

Part II

On Combining Graphs



Giovanni Boccaccio, Decameron, Hamilton 90 Codex.

Capital letters in different sizes and colours remark different levels of the frame stories. Frame stories are just one example of nested content in literature.

4 On Joining Property Graphs

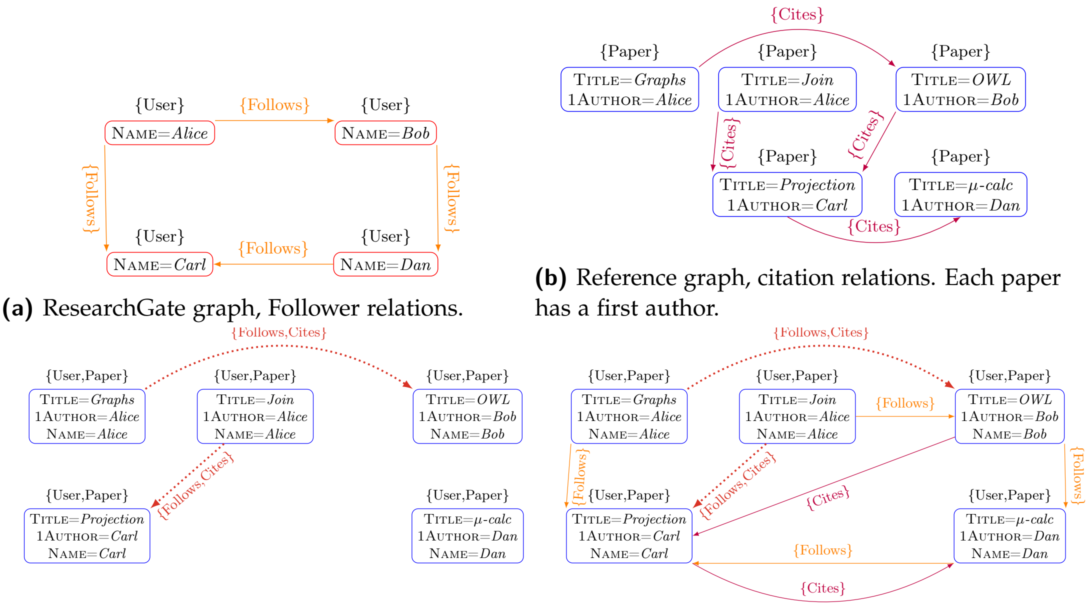
Contents

> *A Cypher query walks into a bar and sees two graphs. It walks up to them and says, "May I join you"?.*
>
> — Restating a well known pun on SQL.

The hight availability of graph data demands for a binary operator combining two distinct graphs into one single graph. Since in the relational model this task is entrusted to join operators, we would expect that similar tasks already appear to be on graph query languages. As described in Section 2.2.2, the join operation is just one of the two operations required for data integration alongside with grouping. While the graph grouping operation has already been defined [JPR17], this graph operation is still missing on current Graph DataBase Management Systems (GDBMS).

Despite the term "join" appearing in graph database literature, such operator could not be used to combine two distinct graphs, as for tables' joins in the relational model. Such joins are *path joins* running over a single graph [ATOR16]: they are used for graph traversal queries [GYQ+12, MSV17] where vertices and edges are considered as relational tables [SFS+15, HG16]. The result of such *path joins* could not be directly used to combine values from different sources (e.g. join two distinct vertices appearing in different graphs alongside with their values), and hence supplementary graph operations are required. The graph integration operation resulting from this combination of two operators does not scale on the large, thus motivating for a specific operator. Such operations require matching traversal paths and then generating new paths from the older ones. Moreover, the term join (SJoin [GCR+17]) has been also recently used to express a link discovery operator between subgraphs of different graph ontologies, where each subgraph represents one single entity alongside with its attributes. Even in this case, the proposed graph join operator does not resemble a binary relational join between graphs, because no new graph is produced as an output, and the only result of such operation is the definition of new correspondences

As for relational databases, they solve common graph queries efficiently, so GDBMS rely either on relational database engines [ATOR16, PLB15, EM09] or on column store databases [SFS+15, BDK+13]. Moreover, relational databases already have efficient implementations



(a) ResearchGate graph, Follower relations.

(b) Reference graph, citation relations. Each paper has a first author.

(c) First query: ResearchGate$\bowtie^{\wedge}_{\texttt{Name=1Author}}$Reference

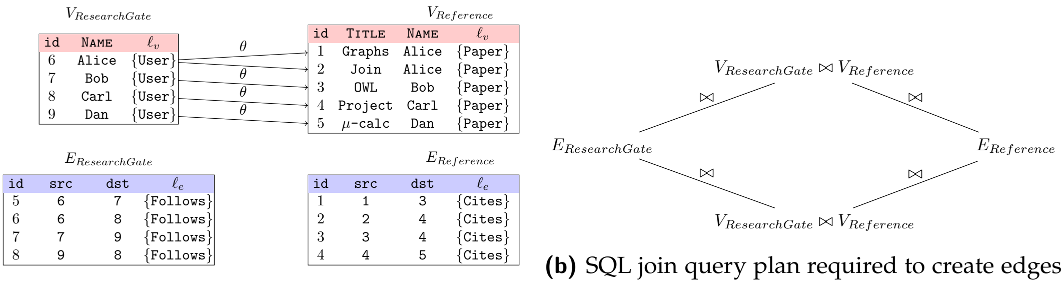
(d) Second query: ResearchGate$\bowtie^{\vee}_{\texttt{Name=1Author}}$Reference

Figure 4.1 Example of a Graph Database for an Enterprise. Dotted edges remark edges shared between the two different joins.

for (equi) join algorithms [SCD16]. We want to show that graph joins over the relational data model are not inefficient. Before all, let us see an example of a graph join query:

▸ **Example 17.** *Consider an on-line service such as* ResearchGate *(Figure 4.1a, or Academia.edu) where researchers can* follow *each others' work, and a* citation *graph (Figure 4.1b). Now we want to "return the paper graph where a paper cites another one iff. the* first author 1Author *of the first paper* follows *the* 1Author *of the second. (Figure 4.1c)". The* ResearchGate *graph does not contain any edge regarding the* references*, while the* Reference *graph does not contain any information pertaining to the follow relations. This demands a join between the two graphs: as a first step we join the vertices together as in the relational model (vertices are considered as tuples using* Name = 1Auth *as a vertex equi-join predicate, θ) and then combine the edges from both graphs. Accordingly to the query formulation, we establish an edge between two joined vertices only if the source has a paper citing the destination, and the user in the source follows the user in the destination.*

Let us now examine the graph join implementation within the relational model: vertices and edges are represented as two relational tables ([SFS$^+$15], Figure 4.2a). In addition to the attributes within the vertices' and the edges' tables, we assume that each row (on both vertices and edges) has an attribute `id` enumerating vertices and edges. Concerning SQL interpretation of such graph join, we first join the vertices (see the records linked by θ lines in Figure 4.2a). Then the edges are computed through the join query provided in Figure 4.2b: the root and the leaves are the result of the θ join between the vertices, while the edges appear as the intermediate nodes. An adjacency list representation of a graph, as the one proposed in the current paper, reduces the joins within the relation solution to one (each vertex and edge is traversed only once), thus reducing the number of required operation to create the resulting graph.



$V_{ResearchGate}$

id	Name	ℓ_v
6	Alice	{User}
7	Bob	{User}
8	Carl	{User}
9	Dan	{User}

$V_{Reference}$

id	Title	Name	ℓ_v
1	Graphs	Alice	{Paper}
2	Join	Alice	{Paper}
3	OWL	Bob	{Paper}
4	Project	Carl	{Paper}
5	μ-calc	Dan	{Paper}

$E_{ResearchGate}$

id	src	dst	ℓ_e
5	6	7	{Follows}
6	6	8	{Follows}
7	7	9	{Follows}
8	9	8	{Follows}

$E_{Reference}$

id	src	dst	ℓ_e
1	1	3	{Cites}
2	2	4	{Cites}
3	3	4	{Cites}
4	4	5	{Cites}

θ
θ
θ
θ
$V_{ResearchGate} \bowtie V_{Reference}$
⋈
⋈
$E_{ResearchGate}$
$E_{Reference}$
⋈
⋈
$V_{ResearchGate} \bowtie V_{Reference}$

(a) Representing the operands' vertices and edges with tables. The θ join for the vertices only involves tables $V_{ResearchGate}$ and $V_{projects}$.

(b) SQL join query plan required to create edges for ResearchGate$\bowtie^{\wedge}_{\texttt{Name=1Author}}$Reference. The leaves acts as the edges' sources while the root as their destinations.

Figure 4.2 Graphically representing the relational join procedure required to evaluate the first query (Figure 4.1c).

Example 17 showed only one possible way to combine the operands' edges, but we can even return edges pertaining to both operands as in the following query: "*For each paper reveal both the direct and the indirect dependencies (either there is a direct paper citation, or one of the authors follows the other one in ResearchGate)*". The resulting graph (Figure 4.1d) has the same vertex set than the previous one, but they differ on the final edges. This implies that our graph join definition must be general enough to allow different edge combinations: we refer to those as **edge semantics**, "**es**" for shorthand.

This chapter provides the following main contributions:

- **(binary) Graph θ-join operator**; we first outline a simple graph model in order to provide a first intuitive definition of a graph join (see our Technical Report [BMM16] outlined in Section 4.2). The operator is both commutative (by swapping the graph operands, the result doesn't change) and associative (it doesn't matter which graph is joined first). Finally, such data model is closed under graph join (i.e. the output of the computation is a graph).
- **Graph Conjunctive Equi-join Algorithm for a specific graph combination task** ("conjunctive", Section 4.4.1): we compare it to its implementation over both graph (SPARQL, Cypher) and relational (SQL) query languages: as a result our solution outperforms the query plan implementation of the other query languages and scales on the large. An example on how to use the same algorithm for less-equal predicate is also prodived (**Graph Conjunctive Less-equal Algorithm**, Section 4.5)
- Graph Joins are then generalized into **left**, **right** and **full** joins, and it is also showed how to extend the first graph join definition to implement such operators (Section 4.6). We show how full graph joins can be used to integrate both graph data and graph schemas into one single intermediate representation.

4.1 Graph Query Languages limitations' on Graph Joins

The reason of comparing our graph join with multiple graph query languages is twofold: we want both to show that graph joins can be represented in different data representations (RDF and Property Graphs), and to detail how our experiments in Section 4.4.2 were performed.

Graph Selection Languages. A first reason why it is impossible to implement graph joins on such languages is that they perform graph query patterns among the edges only one graph at a time. Even though we want to express the edge and vertex match in two distinct graph operands that have to be rewritten in one single graph, such languages do not allow to visit two distinct graph components contemporaneously, that is to check if two graph patterns are matched at the same time, and also they do not support binary predicates between two distinct nodes. At this point suppose that we decide to implement the path join as a binary operation over a graph and the matching graph pattern: even if this interpretation were possible, such languages do not allow to merge into one final result node the query's values with the graph ones, since such languages are only designed to extract specific subgraph from the data graph. The inability of expressing such operator within such query languages discards them from a comparison with our graph join algorithm.

On (Proper) Graph Query Languages. At the time of writing, such graph query languages do not provide a specific keyword to operate the graph join between two graphs. Moreover, all the current graph query languages, except SPARQL, assume that the underlying GDBMS stores only one graph at a time, and hence binary graph join operations are not supported. As a consequence, an operator over one single graph operand must be implemented instead.

Let us now suppose that we want to express a specific θ-join, where the binary predicate θ is fixed, into a (Property) Graph Query Language: in this case we have both *(i)* to specify how a final merged vertex $v \oplus v''$ is obtained from each possible pair of vertices containing different possible attributes $A_1, A_2, \dots, A_n$, and *(ii)* to discard the pair of vertices that do not jointly satisfy the θ predicate and the following *join condition* (that varies upon the different vertices' attributes over the graph):

$$(v \oplus v'')[A_1] = v \wedge (v \oplus v'')[A_2] = v''$$

Please notice that A_1 and A_2 explicitly refer to the final graphs' attributes appearing only on one graph operand. Moreover, for each possible graph join operator, we have to specify which vertices are going to be linked in the final graph and whose nodes are going to have no neighbours.

Among all the possible graph query languages over property graph model, we consider Cypher. An example of the implementation of the an equi-join operator is provided in Figure 4.3: the `CREATE` clause has to be used to generate new vertices and edges from graph patterns extracted through the `MATCH...WHERE` clause, and intermediate results are merged with `UNION ALL`. While current graph query languages allow to express our proposed graph join operator as a combination of the aforementioned operators, our study shows that our specialized graph join algorithm outperforms the evaluation of the graph join with existing graph and relational query languages.

On the other hand, in the case of RDF graph models, we have to discriminate whether vertices either represent entities or values that describe them and, consequently, we have to discriminate between edges representing relations among entities and the ones acting as attributes (when such each links an entity to its associated value expressed as the destination vertex, see Definition 15 on page 74). Even in this case the *join condition* depends upon the specific graphs' schema, that may vary on different RDF graphs. In particular, as showed in Figure 4.4, SPARQL allows to access multiple graph resources through *named graphs* and performs graph traversals one graph at a time through *path joins* [FB09, ACZH10, YLW+13]. At this point the `CONSTRUCT` clause is required if we want to finally combine the traversed paths from both graphs into a resulting graph.

```
MATCH (src1)-[:r]->(dst1),
     (src2)-[:r]->(dst2)
WHERE src1.Organization1=src2.Organization2 AND src1.Year1=src2.Year2
    ↪ AND dst1.Organization1=dst2.Organization2 AND dst1.Year1=dst2.
    ↪ Year2 AND src1.graph=’L’ AND src2.graph=’R’ AND dst1.graph=’L’ AND
    ↪  dst2.graph=’R’
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 ,  MyGraphLabel
    ↪ :”U-”})-[:r]->(:U {Organization1:dst1.Organization1, Organization2
    ↪ :dst2.Organization2 , Year1:dst1.Year1, Year2:dst2.Year2,
    ↪ MyGraphLabel:”U-”}) return p
UNION ALL
MATCH (src1)-[:r]->(u), (src2)-[:r]->(v)
WHERE src1.Organization1=src2.Organization2 AND src1.Year1=src2.Year2
    ↪ AND src1.graph=’L’ AND src2.graph=’R’ AND ((u.Organization1<>v.
    ↪ Organization2 OR u.Year1<>v.Year2))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :”U-”}) return p
UNION ALL
MATCH (src1)-[:r]->(u), (src2)
WHERE src1.Organization1=src2.Organization2 AND src1.Year1=src2.Year2
    ↪ AND src1.graph=’L’ AND src2.graph=’R’ AND (NOT ((src2)-[:r]->()))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :”U-”}) return p
UNION ALL
MATCH (src1), (src2)-[:r]->(v)
WHERE src1.Organization1=src2.Organization2 AND src1.Year1=src2.Year2
    ↪ AND src1.graph=’L’ AND src2.graph=’R’ AND (NOT ((src1)-[:r]->()))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :”U-”}) return p
UNION ALL
MATCH (src1), (src2)
WHERE src1.Organization1=src2.Organization2 AND src1.Year1=src2.Year2
    ↪ AND src1.graph=’L’ AND src2.graph=’R’ AND (NOT ((src2)-[:r]->()))
    ↪ AND (NOT ((src1)-[:r]->()))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :”U-”}) return p
```

Figure 4.3 Cypher implementation for the graph equi-join operator. Please note that one of the limitations of such query language is that each vertex and edge is going to be visited and path-joined more than one time for each pattern where it appears.

```
CONSTRUCT {
        ?newSrc <http://jackbergus.alwaysdata.net/graph> "Result";
              <http://jackbergus.alwaysdata.net/edges/result> ?newDst;
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip1;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org1;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y1;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip2;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org2;
              <http://jackbergus.alwaysdata.net/property/Year2> ?y2.
        ?newDst <http://jackbergus.alwaysdata.net/graph> "Result";
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip3;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org3;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y3;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip4;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org4;
              <http://jackbergus.alwaysdata.net/property/Year2> ?y4.
}
FROM NAMED <leftpath/to/graph>
FROM NAMED <rightpath/to/graph>
WHERE
{
  GRAPH ?g {
                ?src1 <http://jackbergus.alwaysdata.net/property/Id> ?id1;
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip1;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org1;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y1.
        }.
  GRAPH ?h {
                ?src2 <http://jackbergus.alwaysdata.net/property/Id> ?id2;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip2;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org2;
              <http://jackbergus.alwaysdata.net/property/Year2> ?y2.
        }
  filter(?g=<leftpath/to/graph> &&
         ?h=<rightpath/to/graph> &&
         ( ?org1 = ?org2 ) && ( ?y1 =  ?y2 ))

  BIND (URI(CONCAT("http://jackbergus.alwaysdata.net/values/",?id1,"-",?id2)) AS ?newSrc)

  OPTIONAL {
                GRAPH ?g {
                        ?src1 <http://jackbergus.alwaysdata.net/edges/edge> ?dst1.
                        ?dst1 <http://jackbergus.alwaysdata.net/property/Id> ?id3;
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip3;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org3;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y3.
                }.
                GRAPH ?h {
                        ?src2 <http://jackbergus.alwaysdata.net/edges/edge> ?dst2.
                        ?dst2 <http://jackbergus.alwaysdata.net/property/Id> ?id4;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip4;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org4;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y4.
                }
                FILTER ( ( ?org3 = ?org4 ) && ( ?y3 = ?y4 ) )
                BIND (URI(CONCAT("http://jackbergus.alwaysdata.net/values/",?id3,"-",?id4))
                     ↪ AS ?newDst)
        }
}
```

Figure 4.4 SPARQL implementation for the graph equi-join operator. Please note that while the SPARQl engine can optimize the graph traversal tasks, the creation of a new graph as a result is not included in the optimization steps.

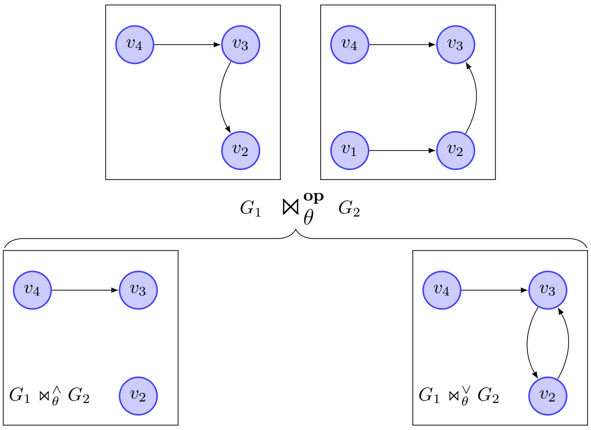

v_4
v_3
v_2
v_4
v_3
v_1
v_2
$G_1 \bowtie_{\theta}^{\mathbf{op}} G_2$
v_4
v_3
$G_1 \bowtie_{\theta}^{\wedge} G_2$
v_2
v_4
v_3
$G_1 \bowtie_{\theta}^{\vee} G_2$
v_2

Figure 4.5 Given two graph with vertices with same id, hence sharing the same value, the graph conjunctive join extracts the common pattern, while the disjunctive join retrieves at least one edge shared among the matched nodes.

Consequently, for the two following conditions current graph query languages do not support graph joins as a primitive operator.

1. An explicit graph join operator is missing in current graph database languages.
2. Even if we hold fixed the θ binary property, each time that the underlying graph schema changes we have to rewrite the join query every time, either because we have to specify how to merge the nodes and how to create final edges, or because we have to re-write the *join condition*.

4.2 Graph Data Model

Differently from the previously mentioned relational model, we now provide a data model allowing an explicit representation of tuples, which may appear more than once within a single set (e.g., vertex or edge set.). Still, we decide to embed the relational model on property graphs, so that the standard operators' properties from the relational model can be inherited.

We model the vertices' and edges' set as **multisets (of tuples)** S of elements s_i, where s_i unequivocally identifies the i-th occurrence of a tuple s in S. Each **tuple** associates to each attribute a value: it is a function $A \mapsto \mathcal{V} \cup \{\texttt{NULL}\}$ mapping each attribute in A to either a value in $\mathcal{V}$ or NULL (ε is the empty tuple). We slightly change the previous property graph definitions in order to ease the join definition between vertices and edges as later on required by the graph join:

▸ **Definition 21** (Property Graph)**.** *A **property graph** is a tuple $G = (V, E, \Sigma_v, \Sigma_e, A_v, A_e, \lambda, \ell_v, \ell_e)$ where **(a)** V is a multiset of nodes, **(b)** E is a multiset of edges, **(c)** Σ_v is a set of node labels, **(d)** Σ_e is a set of edge labels, **(e)** A_v is a set of node attributes, **(f)** A_e is a set of edge attributes, **(g)** $\lambda\colon E \to V \times V$ is a function assigning node pairs to edges, **(h)** $\ell_v\colon V \to \mathcal{P}(\Sigma_v)$ is a function assigning a set of labels to nodes, and **(i)** $\ell_e\colon E \to \mathcal{P}(\Sigma_e)$ is a function assigning a set of labels to edges.* ◂

Given that the standard property graph model is unable to model graphs and databases in a similar representation, we must provide the following definition:

▸ **Definition 22** (Graph Database). *A **graph database** is a collection of n distinct property graphs $\{G_1, \dots, G_n\}$ represented as a single property graph $\mathcal{D}$ with n distinct connected components. From now on we refer to each component simply as **graph**. Each graph is identified by two functions: $\mathcal{V}\colon \{1, \dots, n\} \mapsto \mathcal{P}(V)$ determining the vertices $\mathcal{V}(i)$ of the i-th graph and $\mathcal{E}\colon \{1, \dots, n\} \mapsto \mathcal{P}(E)$ determining the edges $\mathcal{E}(i)$ of the i-th graph.* ◂

▸ **Example 18.** *Two edges e_i and f_j come from two distinct graphs, respectively G_a and G_b, within the same graph database $\mathcal{D}$. Edge e_i connects vertex u_h to v_k ($\lambda(e_i) = (u_h, v_k)$), while f_j connects u'_h to v'_k ($\lambda(f_j) = (u'_h, v'_k)$). Such edges store only the following values:*

$$e_i(\textsc{Time}) = 12{:}04, \quad f_j(\textsc{Day}) = \textit{Mon}$$

and have the following labels:

$$\ell_e(e_i) = \{\,\text{Follow}\,\}, \quad \ell_e(f_j) = \{\,\text{FriendOf}\,\}$$

For the multiset θ-join, we need a function $\oplus$ combining two tuples for the relational join operator over multisets, where $r_i \oplus t_j$ is a valid multiset element $(r \oplus s)_{i \oplus j}$ and $i \oplus j$ maps each integer pair (i, j) to a single number. If we define $\oplus$ as a linear function (that is for each function H, $H(e_i \oplus f_j) = H(e_i) \oplus H(f_j)$), the θ-join also induces the definition of ℓ_v, ℓ_e and λ for the joined tuples. As a consequence, $\oplus$ must be overloaded for each possible expected output from H. Such function is defined as follows:

▸ **Definition 23** (Concatenation). *$\oplus : A \times A \mapsto A$ is a lazy evaluated **concatenation** function between two operands of type A returning an element of the same type, A. The concatenation function is a linear function such that, given any function H with $dom(H) = A$, $H(u \oplus v) = H(u) \oplus H(v)$. $\oplus$ is defined for the following A-s:*

- ***sets**: it performs the union of the two sets:* $S \oplus S' \overset{def}{=} S \cup S'$
- ***integers**: it returns the dovetail number associating to each pair of integers an unique integer:* $i \oplus j \overset{def}{=} \sum_{k=0}^{i+j} k + i$
- ***functions**: given a function $f : A \mapsto B$ and $g : C \mapsto D$, $f \oplus g$ is the overriding of f by g returning $g(x)$ if $x \in \operatorname{dom}(g)$, and $f(x)$ if $x \in \operatorname{dom}(f)$. NULL is returned otherwise. Such function concatenation are used in joins when $\forall x \in A \cap C. f(x) = g(x)$.*
- ***pairs**: given two pairs (u, v) and (u', v'), then the pair concatenation is defined as the pairwise concatenation of each element, that is $(u, v) \oplus (u', v') \overset{def}{=} (u \oplus u', v \oplus v')$. Elements belonging to multisets are represented as pairs of elements and integers, and hence $s_i \oplus t_j \overset{def}{=} (s \oplus t)_{i \oplus j}$.*

◂

After providing the definition of the concatenation function, we can provide the graph join definition as follows:

▸ **Definition 24** (θ-Join). *Given two (multiset) tables R and S over a set of attributes A_1 and A_2, the **θ-join** $R \bowtie_\theta S$ [ACPT99, ACPT09] is defined as follows:*

$$R \bowtie_\theta S = \{r_i \oplus s_j \mid r_i \in R, s_j \in S, \theta(r_i, s_j), (r_i \oplus s_j)(A_1) = r_i, (r_i \oplus s_j)(A_2) = s_j\}$$

where $(t \oplus t')(A_i)$ denotes the projection of the tuple $t \oplus t'$ over A_i. If θ is the always true predicate, θ can be omitted and, when also $A_1 \cap A_2 = \varnothing$, we have a cartesian product. ◂

▸ **Example 18** (continuing from p. 96). *Suppose now that the edge $e_i \oplus f_j$ comes from a graph join where edges from G_a are joined to the ones in G_b in a resulting graph, where also vertices $u_h \oplus u'_h$ and $v_k \oplus v'_k$ appear. So:*

$$(e_i \oplus f_j)(\textsc{Time}) = 12{:}04, \quad (e_i \oplus f_j)(\textsc{Day}) = Mon$$

By $\oplus$'s linearity, we have that the labels are merged:

$$\ell_e(e_i \oplus f_j) = \ell_e(e_i) \oplus \ell_e(f_j) = \{\text{Follow}\} \oplus \{\text{FriendOf}\} = \{\text{Follow}, \text{FriendOf}\}$$

And the result's vertices are updated accordingly:

$$\lambda(e_i \oplus f_j) = \lambda(e_i) \oplus \lambda(f_j) = (u_h, v_k) \oplus (u'_u, v'_k) = (u_h \oplus u'_h, v_k \oplus v'_k)$$

Since all the relevant informations are stored in the graph database, we represent the graph as the set of the minimum information required for the join operation.

▸ **Definition 25** (Graph). *The i-th **graph** of a graph database $\mathcal{D}$ is a tuple $G_i = (\mathcal{V}(i), \mathcal{E}(i), A_v^i, A_e^i)$, where $\mathcal{V}(i)$ is a multiset of vertices and $\mathcal{E}(i)$ is a multiset of edges. Furthermore, A_v^i is a set of attributes $a \in A_v^i$ s.t. there is at least one vertex $v_j \in \mathcal{V}(i)$ having $v_i(a) \neq$ `NULL`; A_e^i is a set of attributes $a' \in A_e^i$ s.t. there is at least one edge $e_k \in \mathcal{E}(i)$ having $e_k(a') \neq$ `NULL`.* ◂

4.3 Graph θ-Joins

At the time of writing, the only field where graph joins where effectively discussed is Discrete Mathematics. In this field such operations are defined over either on finite graphs or on finite graphs with cycles, and are named *graph products* [HIK11]. As the name suggests, every graph product of two graphs, e.g. $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, produces a graph whose vertex set is defined as $V_1 \times V_2$, while the edge set changes accordingly to the different graph product definition. Consequently the Kroneker Graph Product [Wei62] is defined as follows:

$$G_1 \times G_2 = (V_1 \times V_2, \{\, ((g,h),(g',h')) \in V_1 \times V_2 \mid (g,g') \in E_1, (h,h') \in E_2 \,\})$$

while the *cartesian graph product* [IP07] is defined as follows:

$$G_1 \,\square\, G_2 = (V_1 \times V_2, \{\, ((g,h),(g',h')) \in V_1 \times V_2 \mid (g = g', (h,h') \in E_2) \vee (h = h', (g,g') \in E_1) \,\})$$

Please observe that this definition creates a new vertex which is a pair of vertices: hereby such operation is defined differently from the relational algebra's cartesian product, where the two vertices are merged. As a consequence, such graph products admit commutativity and associativity properties only up to graph isomorphism. Other graph products are *lexicographic product* and *strong product* [HIK11, IK00]. Recently, a Kronecker Product for edge uncertainty was also provided [ZY17], even though no information on how to combine the data associated to either vertices or edges was provided.

Consequently, even our graph join is based on the combination of vertices and edges: $G_a \bowtie_\theta^{\mathbf{es}} G_b$ expresses the join of graph G_a with G_b where *(i)* we first use a relational θ-join among the vertices, and then *(ii)* we combine the edges using an appropriate user-determined edge semantics, **es**. This modularity is similar to the operators previously described in graph theory literature, where instead of a join between vertices they have a cross product, and different semantics are expressed as different graph products. We now provide the graph join definition:

▸ **Definition 26** (Graph θ-Join). *Given two graphs $G_a = (V, E, A_v, A_e)$ and $G_b = (V', E', A'_v, A'_e)$, a **graph θ-join** is defined as follows:*

$$G_a \Join^{\mathbf{es}}_{\theta} G_b = (V \Join_\theta V', E_{\mathbf{es}}, A_v \cup A'_v, A_e \cup A'_e)$$

*where θ is a binary predicate over the vertices and $\Join_\theta$ the θ-join (Definition 24) among the vertices, and $E_{\mathbf{es}}$ is a subset of all the possible edges linking the vertices in $V \Join_\theta V'$ expressed with the **es** semantics.* ◂

Given that graph join returns a property graph like the graphs in input, property graphs are closed under the graph join operator via the definition of $\oplus$ for the multiset θ-join. Moreover, this allows a first step towards the edge result combination as required by some data integration scenarios. For example, The result of the join between two graphs, ResearchGate (Figure 4.1a) and References (Figure 4.1b), produces the same set of vertices regardless of the edge semantics of choice. On the other hand, edges among the resulting vertices change according to the edge semantics. In the first one (Figure 4.1c) we combine edges appearing in both graphs and linking vertices that appear combined in the resulting graph. We have a **Conjunctive Join**, that in graph theory is known as *Kronecker graph product* [Wei62, HIK11]. In this case $E_{\mathbf{es}}$ is defined with the "$\wedge$" **es** semantics as an edge join $E_\wedge = E \Join_{\Theta_\wedge} E'$, where the $\Theta_\wedge$ predicate is the following one:

$$\Theta_\wedge(e_h, e'_k) = (e_h \in E \wedge e'_k \in E') \wedge \lambda(e_h \oplus e'_k) \in (V \Join_\theta V')^2 \tag{4.1}$$

We can also define a **disjunctive** semantics (Figure 4.1d), having "$\vee$" as **es**. In this case we want edges appearing either in the first or in the second operand. This means that two vertices, $u_h \oplus u'_h$ and $v_k \oplus v'_k$, could have a resulting edge $e_i \oplus \varepsilon'_j$ even if only $\lambda(e_i) = (u_h, v_k)$ appears in the first operand[1] and ε' is a "fresh" empty edge $\lambda(\varepsilon'_j) = (u'_h, v'_k)$ not appearing in G_b such that $\lambda(e_a \oplus \varepsilon'_b) = (u_h \oplus u'_h, v_k \oplus v'_k)$. Consequently the disjunctive join can be represented as a *full outer join*, where the edges either match in the conjunctive semantics, or appear in the two distinct graph operands:

$$\begin{aligned} E_\vee = E \mathbin{⟗}_{\Theta_\wedge} E' \quad & \cup \{(e_i \oplus \varepsilon_j) | \varepsilon_j, \left(\forall e' \in E'.\neg\Theta_\wedge(e_i, e')\right), \ell_e(\varepsilon_j) = \varnothing,\ \lambda(e_i) \oplus (v, v') \in (V \Join_\theta V')^2\} \\ & \cup \{(\varepsilon_i \oplus e_j) | \varepsilon_i, \left(\forall e \in E.\neg\Theta_\wedge(e, e_j)\right), \ell_e(\varepsilon_i) = \varnothing,\ (u, u') \oplus \lambda(e_j) \in (V \Join_\theta V')^2\} \\ = E \mathbin{⟗}_{\Theta_\wedge} E' & \end{aligned}$$

4.3.1 Graph Join properties

With this section we want to discuss some properties of graph joins that allow to analyse their scalability with respect to the generalization of such joins to multiple graphs. We could first check that the proposed graph θ-join is closed under composition: it takes two (property) graphs and returns a property graph as an output by construction. We discuss the commutativity and the associativity for graph joins in each semantics.

▸ **Lemma 1** (Join Commutativity). *Given two graphs G and G' from the same graph database $\mathcal{D}$ and a symmetric binary predicate θ, then we have $G \Join^\wedge_\theta G' \equiv G' \Join^\wedge_{\theta^{-1}} G$ for the conjunctive semantics and $G \Join^\vee_\theta G' \equiv G' \Join^\vee_{\theta^{-1}} G$ for the disjunctive one.*

[1] The statement addressing the edges in the second operand that do not bond with the ones in the other graph is expressed as follows: $\forall e' \in E'.\neg\Theta_\wedge(e_i, e')$

Proof. We first choose θ^{-1} as the inverse predicate of θ, such that $\theta^{-1}(b,a) \Leftrightarrow \theta(a,b)$. For the conjunctive semantics, we have that $G \bowtie_\theta^\wedge G'$ is:

$$(V \bowtie_\theta V', E \bowtie_{\Theta_\wedge} E', A_v \cup A'_v, A_e \cup A'_e, \mathcal{V}_v \cup \mathcal{V}'_v, \mathcal{V}_e \cup \mathcal{V}'_e)$$

Since we have that $V \bowtie_\theta V' = V' \bowtie_{\theta^{-1}} V$ and $E \bowtie_{\Theta_\wedge} E' = E' \bowtie_{\Theta_\wedge} E$, then we have that the graph join is equivalent to $G' \bowtie_{\theta^{-1}}^\wedge G$.

This is proved because the relational join between vertices and edges is a commutative operator [Rö94], and predicate $\Theta_\wedge$ is symmetric when either E and E' or E' and E are joined. A similar proof could be carried out for the disjunctive semantics, since it is in the form:

$$\begin{aligned}
E \mathbin{⟗}_{\Theta_\wedge} E' \quad & \cup\{(e_i \oplus \varepsilon_j)|\varepsilon_j, \big(\forall e' \in E'.\neg\Theta_\wedge(e_i,e')\big), \ell_e(\varepsilon_j) = \varnothing,\ \lambda(e_i) \oplus (v,v') \in (V \bowtie_\theta V')^2\} \\
& \cup\{(\varepsilon_i \oplus e_j)|\varepsilon_i, \big(\forall e \in E.\neg\Theta_\wedge(e,e_j)\big), \ell_e(\varepsilon_i) = \varnothing,\ (u,u') \oplus \lambda(e_j) \in (V \bowtie_\theta V')^2\} \\
= E' \mathbin{⟗}_{\Theta_\wedge} E \quad & \cup\{(e_j \oplus \varepsilon_i)|\varepsilon_i, \big(\forall e \in E.\neg\Theta_\wedge(e,e_j)\big), \ell_e(\varepsilon_i) = \varnothing,\ (u',u) \oplus \lambda(e_j) \in (V' \bowtie_\theta V)^2\} \\
& \{(\varepsilon_j \oplus e_i)|\varepsilon_j, \big(\forall e' \in E'.\neg\Theta_\wedge(e_i,e')\big), \ell_e(\varepsilon_j) = \varnothing,\ \lambda(e_i) \oplus (v',v) \in (V' \bowtie_\theta V)^2\}
\end{aligned}$$

This equation is true because the relational join is symmetric as the set union and the $\oplus$ operator with the null tuple ε_i. ◂

The following corollary strengthens the previous result: it shows that join commutativity implies having two resulting graphs where both vertices and edges have the same labels, and the same edges link the same vertices.

▸ **Corollary 1** (Commutativity for λ, ℓ_v and ℓ_e)**.** *For each vertex $v_i \oplus v'_j$ from the vertex set $V \bowtie_\theta V'$ from $G \bowtie_\theta^\wedge G'$ and the corresponding equivalent vertex $v'_j \oplus v_i$ in $V' \bowtie_\theta V$ from $G' \bowtie_\theta^\wedge G$ ($v_i \oplus v'_j = v'_j \oplus v_i$), we have that both vertices have the same label set.*

For the conjunctive semantics, for each edge $e_h \oplus e'_k$ from the edge set $E \bowtie_{\Theta_\wedge} E'$ from $G \bowtie_\theta^\wedge G'$ and the corresponding equivalent edge $e'_k \oplus e_h$ in $E' \bowtie_{\Theta_\wedge} E$ from $G' \bowtie_\theta^\wedge G$ ($e_h \oplus e'_k = e'_k \oplus e_h$), we have that both edges have the same label set and link the same equivalent vertices. This statement also applies for the disjunctive semantics.

Proof. This corollary is proved by the linearity of $\oplus$. Regarding the vertex labelling, we have that the labelling provided by the result of the two commutated joins is the same by the commutativity of the set union operator:

$$\begin{aligned}
\ell_v(u_i \oplus u'_j) = \ell_v(u_i) \oplus \ell_v(u'_j) &= \ell_v(u_i) \cup \ell_v(u'_j) = \\
&= \ell_v(u'_j) \cup \ell_v(u_i) = \ell_v(u'_j) \oplus \ell_v(u_i) = \\
&= \ell_v(u'_j \oplus u_i)
\end{aligned}$$

Regarding the conjunctive semantics, the proof of $\ell_e(e_h \oplus e'_k) = \ell_e(e'_k \oplus e_h)$ is similar, by using the set union's commutativity. We prove that equivalent edges link equivalent vertices:

$$\begin{aligned}
\lambda_{E_\wedge}(e_h \oplus e'_k) = \lambda_E(e_h) \oplus \lambda_{E'}(e'_k) &= (u_i, v_j) \oplus (u'_l, v'_m) = \\
&= (u_i \oplus u'_l, v_j \oplus v'_m) = (u'_l \oplus u_i, v'_m \oplus v_j)
\end{aligned}$$

$$\lambda_{E_\wedge}(e'_k \oplus e_h) = \lambda_{E'}(e'_k) \oplus \lambda_{E'}(e'_k) = (u'_l \oplus u_i, v'_m \oplus v_j)$$

The proofs for the disjunctive semantics are the same. ◂

Since the relational algebra θ-join operator satisfies associativity [Rö94], we could carry out a similar proof for join associativity:

▸ **Lemma 2** (Join Associativity). *Given three graphs G, G' and G'' from the same graph database $\mathcal{D}$ and a symmetric binary predicate θ, then we have $G \bowtie^{\wedge}_{\theta_1\wedge\theta_\alpha} (G' \bowtie^{\wedge}_{\theta_2} G'') = (G \bowtie^{\wedge}_{\theta_1} G') \bowtie^{\wedge}_{\theta_\alpha\wedge\theta_2} G''$ for the conjunctive semantics and $G \bowtie^{\vee}_{\theta_1\wedge\theta_\alpha} (G' \bowtie^{\vee}_{\theta_2} G'') = (G \bowtie^{\vee}_{\theta_1} G') \bowtie^{\vee}_{\theta_\alpha\wedge\theta_2} G''$ for the disjunctive one.*

Proof. Since we have that the usual θ-relational joins are associative as outlined by the following equivalence:

$$(A \bowtie_{\theta_1} B) \bowtie_{\theta_\alpha\wedge\theta_2} C = A \bowtie_{\theta_1\wedge\theta_\alpha} (B \bowtie_{\theta_2} C)$$

then, we have that the relational θ-joins among the edges are associative too, as well as the theta joins among the edges. Hereby, the join between the graphs is associative. ◂

Similarly to the graph join's commutativity, we can strengthen the result for the join associativity with the following corollary:

▸ **Corollary 2** (Associativity for λ, ℓ_v and ℓ_e). *For each vertex $v_i \oplus (v'_j \oplus v''_k)$ from the vertex set $V \bowtie^{\wedge}_{\theta_1\wedge\theta_\alpha} (V' \bowtie^{\wedge}_{\theta_2} V'')$ from $G \bowtie^{\wedge}_{\theta_1\wedge\theta_\alpha} (G' \bowtie^{\wedge}_{\theta_2} G'')$ and the corresponding equivalent vertex $(v_i \oplus v'_j) \oplus v''_k$ in $V' \bowtie_\theta V$ from $(G \bowtie^{\wedge}_{\theta_1} G') \bowtie^{\wedge}_{\theta_\alpha\wedge\theta_2} G''$ ($v_i \oplus (v'_j \oplus v''_k) = (v_i \oplus v'_j) \oplus v''_k$), we have that both vertices have the same label set.*

For the conjunctive semantics, for each edge $e_h \oplus (e'_k \oplus e''_t)$ from the edge set $E \bowtie_{\Theta_\wedge} (E' \bowtie_{\Theta_\wedge} E'')$ from $G \bowtie^{\wedge}_{\theta_1\wedge\theta_\alpha} (G' \bowtie^{\wedge}_{\theta_2} G'')$ and the corresponding equivalent edge $(e_h \oplus e'_k) \oplus e''_t$ from $(G \bowtie^{\wedge}_{\theta_1} G') \bowtie^{\wedge}_{\theta_\alpha\wedge\theta_2} G''$, we have that both edges have the same label set and link the same equivalent vertices. This statement also applies for the disjunctive semantics.

Proof. This corollary is proved by the linearity of $\oplus$. Regarding the vertex labelling, we have that the labelling provided by the result of the two commutated joins is the same by the associativity of the set union operator:

$$\begin{aligned}\ell_v(u_i \oplus (u'_j \oplus u''_k)) &= \ell_v(u_i) \oplus \ell_v(u'_j \oplus u''_k) = \ell_v(u_i) \cup \ell_v(u'_j) \cup \ell_v(u''_k) = \\ &= \ell_v(u_i \oplus u'_j) \oplus \ell_v(u''_k) = \ell_v((u_i \oplus u'_j) \oplus u''_k)\end{aligned}$$

Regarding the conjunctive semantics, the proof of $e_h \oplus (e'_k \oplus e''_t) = (e_h \oplus e'_k) \oplus e''_t$ is similar, by using the set union's associativity. We prove that equivalent edges link equivalent vertices:

$$\begin{aligned}\lambda_{E_\wedge}((e_h \oplus e'_k) \oplus e''_t) &= \lambda_E(e_h \oplus e'_k) \oplus \lambda_{E'}(e''_t) = (u_i \oplus u'_l, v_j \oplus v'_m) \oplus (u''_n, u''_p) = \\ &= (u_i \oplus u'_l \oplus u''_n, v_j \oplus v'_m \oplus u''_p) = \lambda_{E_\wedge}(e_h \oplus (e'_k \oplus e''_t))\end{aligned}$$

The proofs for the disjunctive semantics are the same. ◂

These properties for the graph joins prepare to a scalable implementation of multi-way graph joins: we could start to perform such joins from the smallest graph up to the greatest graph, such that the number of bucket comparisons is reduced.

4.4 Graph Conjunctive Equi-Joins

In the present section we're going to first describe the graph equi-join for θ conjunctive equijoins, and then introduce the secondary memory graph representation used by the presented algorithm (Section 4.4.1). In Section 4.4.2 we're going to compare our proposed algorithm (GCEA) to both graph database libraries (Section 4.4.2.1 on page 105), on top of which the same algorithm is implemented and evaluated, and graph query languages on top of specific graph databases (Section 4.4.2.2 on page 107), thus comparing the efficiency of our algorithm to their query plan evaluation.

4.4.1 Algorithm and Data Structure

We now outline our algorithm, GCEA, for θ equijoin predicates, involving an equivalence between attributes or a conjunction of such equivalences. We also suppose that both graph operands are always stored within the same graph database, and that the query itself will be provided to be read inside the same database environment. The query result is read only

Algorithm II.1 Graph Conjunctive EquiJoin Algorithm (GCEA)

```
procedure ConjunctiveJoin($G, G', \theta$)
    $hashFunction$ = generateHash($\theta$);
    $omap_1$ = OperandPartitioning($G$,$hashFunction$)
    $omap_2$ = OperandPartitioning($G'$,$hashFunction$)
    $\overline{G}_1$ = SerializeOperand($G$,$omap_1$)
    $\overline{G}_2$ = SerializeOperand($G'$,$omap_2$)
      return PartitionHashJoin($\overline{G}_1, \overline{G}_2, \theta$)

procedure SerializeOperand($G$,omap):
    File $VertexIndex$ = Open();
    $VertexVals$= Open(), $HashOffset$= Open();
    ulong $offset$ = $HashOffset$ = 0;
    for each $h$ ∈Keys(omap) do                    ▷ Ordered maps have ordered keys.
        $HashOffset$.Write({$h$,$HashOffset$});
        for each $id$ ∈omap[$h$] do
            $v = G.V[id]$;
            $v$.hash = $h$; $v$.offset = $VertexVals$;
            $VertexIndex$.Write({$v$.id, $h$, $offset$});
            ulong offsetNext = $VA$.Write(serialize(v));
            $offset$+=offsetNext; $HashOffset$+=offsetNext;
    return ($VertexIndex$,$VertexVals$,$HashOffset$,$G.A_v$,$G.A_e$)

procedure PartitionHashJoin($G_1, G_2, \theta$):
    $\theta'(u,u') := \theta(u,v) \wedge (u \oplus u')(A_v) = u \wedge (u \oplus u')(A'_v) = u'$;
    $\Theta'(e,e') := (e \oplus e')(A_e) = e \wedge (e \oplus e')(A'_e) = e'$
    $HI$ = IntersectHashes($HashOffset_1$,$HashOffset_2$).iterator();
    File $AdjFile$ = Open();
    while $HI$.hasNext() do
        $h = HI$.next();
        for each $u \in VertexVals_1[h.\text{offset}_1]$, $u' \in VertexVals_2[h.\text{offset}_2]$ do
            if $\theta'(u,u')$ then
                $AdjFile$.Write(V={$u \oplus u'$},)
                $HIout$ = IntersectHashes($out_V(u)$,$out_{V'}(u')$).iterator();
                while $HIout$.hasNext() do
                    $h_{out} = HIout$.next();        ▷ Offsets refer to the blocks for u and $u'$ outgoing edges
                    for each edge $e \in out_V(u)[h_{out}.\text{offset}_1]$, $e' \in out_{V'}(u')[h_{out}.\text{offset}_2]$ do
                        if $\theta'(e.\text{outv}, e'.\text{outv})$ and $\Theta'(e,e')$ then
                            $AdjFile$.Write(E={$e \oplus e'$})
```

as in other graph query languages (SPARQL, SQL for databases) and does not correspond to a "materialized view". Therefore, the result of the graph query itself can postpone the creation of a complete property graph output (that is, the complete attribute-value and label information for our graphs).

The starting point of each data integration procedure is the identification of similar pieces of data occurring into the to-be-merged operands. In our case, the graph integration task starts from the identification of the to-be-joined vertices and, after that, we have to decide how to combine the outgoing edges. The most intuitive way to match similar data content is to use equivalence predicates.

This specific predicate choice was driven by the fact that the most performant and implemented relational database join is the equi-join [SCD16], and that the result of relational databases' queries already produces join indices [Dit16] where only the left and right operand indices are kept in the final result, so that the relational tables' pieces of informations are kept in primary memory purely as indices. Please also note that not all the graph join's equijoin predicates may be optimized[2] and consequently, in some cases we must undoubtedly pay the computational price of the cartesian product. Moreover, we provide an implementation for conjunctive semantics, since this task is more prone to be optimized than the disjunctive one. Algorithm II.1 for GCEA consists in three parts: *(i)* vertex partitioning (bucketing) through an hashing function (OperandPartitioning) *(ii)* graph serialization on secondary memory (SerializeOperand), and *(iii)* actual join algorithm over the graphs' buckets (PartitionHashJoin). Relational partition hash-join undergo the same phases, even if relational algorithms do not deal with outgoing edges (lines 31-35). We allow vertices with replicated values as in current graph databases implementations (such as Titan and Neo4J). Consequently, *id*s enumerate the vertices within a single graph.

As a first step, the hashing function h is inferred from θ (line 2): if $\theta(u,v)$ is a binary predicate between distinct attributes from u and v, then h is defined as a linear combination of hash functions over the attributes of either u or v. When no h could be inferred from θ, then h is a constant function.

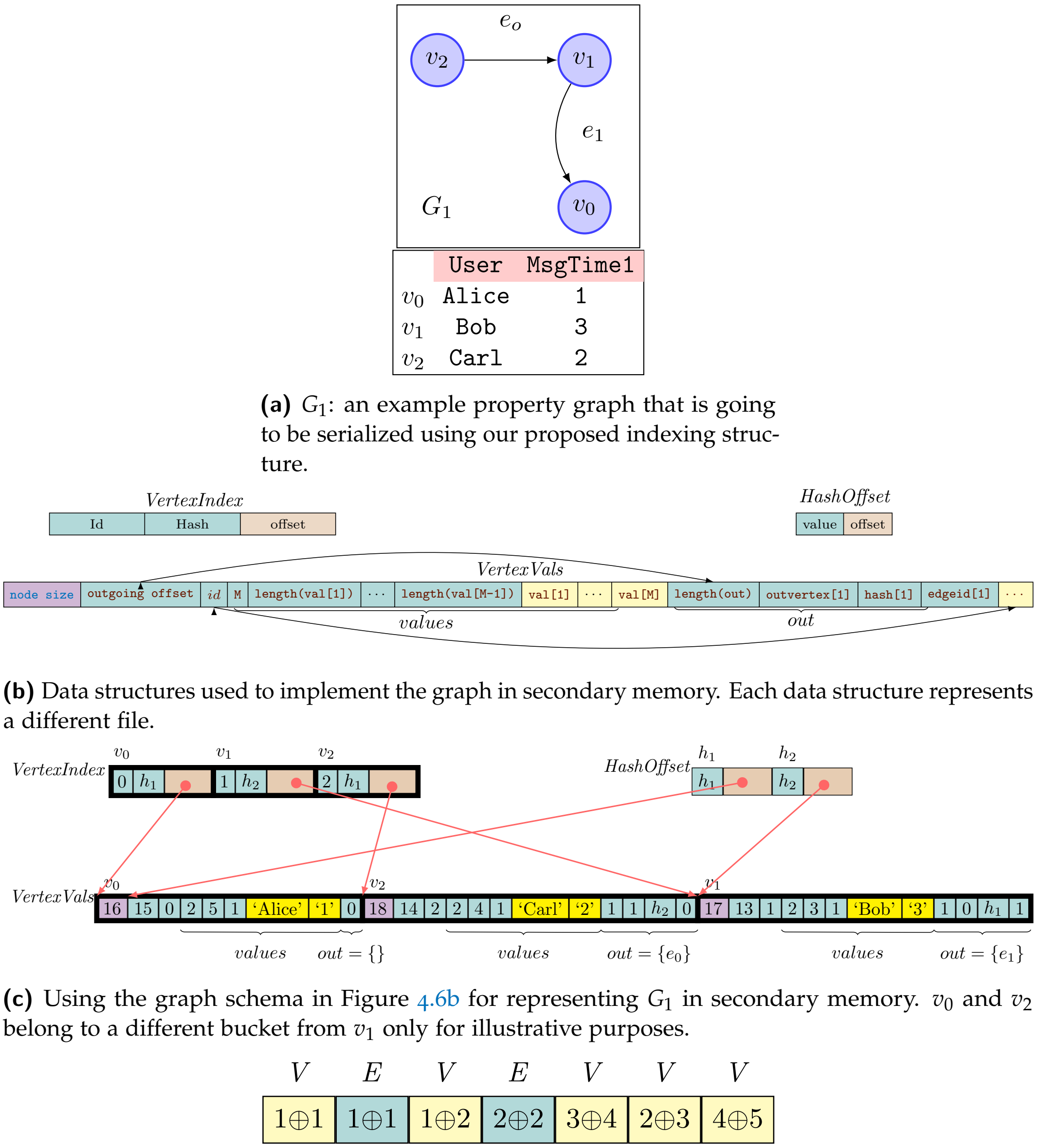
OperandPartitioning performs a vertex bucketing in main memory: its outcome is an ordered map, where each vertex v is stored in a collection $omap[h(v)]$, where h is the aforementioned hashing function. For each operand G_i, the $omap_i$ construction takes at most $\sum_{j=0}^{|\mathcal{V}(i)|}\log(j)$ time, where $|\mathcal{V}(i)|$ is the multiset vertex size. Such time complexity is bounded by $|\mathcal{V}(i)| \leq \sum_{j=0}^{|\mathcal{V}(i)|}\log(j) < |\mathcal{V}(i)|^2$ where $|\mathcal{V}(i)| \gg 1$.

SerializeOperand stores the operand in secondary memory: both buckets (line 12) and vertices (line 14) are already sorted by hash value, and hence such data structures are accessed linearly. Figure 4.6c depicts a serialized representation of the graph in Figure 4.6a: all the labels and the edge values are not serialized but are still accessible through the original graph G via *id*. Moreover, such representation provides a adjacency list representation of graphs, that has been already proved to be graph traversal efficient, even in distributed computation contexts [LBO⁺15]. Buckets are represented by *HashOffset*

[2] Consider two relations $R(A_1, A_2)$ and $S(A_3, A_4)$ within the relational model; we want to θ-join them using the following predicate:

$$\theta(r,s) = r[A_1] - s[A_4] = r[A_2] \cdot s[A_3]$$

Since this predicate requires performing arithmetical operations between attributes appearing in distinct relations ($r[A_1] - s[A_4]$) which evaluation requires the computation of all the $R \times S$ tuples, such predicate cannot benefit from a preliminary hashing and bucketing step.



(a) G_1: an example property graph that is going to be serialized using our proposed indexing structure.

(b) Data structures used to implement the graph in secondary memory. Each data structure represents a different file.

(c) Using the graph schema in Figure 4.6b for representing G_1 in secondary memory. v_0 and v_2 belong to a different bucket from v_1 only for illustrative purposes.

(d) Representing the serialization of the join ResearchGate$\bowtie^{\wedge}_{\texttt{Name=1Author}}$Reference depicted in Figure 4.1c on page 90.

Figure 4.6 *Graph representation in secondary memory.*

providing both the bucket value and the pointer to the first vertex of the bucket stored in *VertexVals*. *VertexVals* stores vertices alongside with their adjacency list, where vertices are sorted by hash value and are represented by *id* and hash value. *VertexIndex* allows to find the vertices stored in *VertexVals* in constant time: each record is ordered by vertex *id*, has a constant size and contains the pointer to where the vertex data is stored in *VertexVals*. Even the outgoing edges are stored by the destination vertex's hash value. Given k_i the size of Keys(omap$_i$), this phase takes $3k_i + |G_i|$ time, where $2k_i$ is the omap visit cost, k_i is the omap serialization as *HashOffset* and $|G_i|$ is the time to serialize the graph as VA_i.

The last step performs the actual conjunctive join over the serialized graph (PartitionHashJoin): the data structure is accessed from secondary memory through memory mapping. Line 24 prepares the intersection: while performing a linear scan over the buckets, the HI iterator checks if both operands have a bucket with the same hash value (line 26), then the common hash value is extracted (line 27) and the two buckets accessed (line 28), then the composition $u \oplus u'$ between the vertices is performed (line 30). Next, differently from the relational join, the adjacent vertices for both operands are visited. Similarly to line 24, the hash-sorted edges induce a bucketing (line 31), and then we check if the destination vertices meet the join conditions alongside with the to-be-joined edges (line 35). Please note that, as stated out in Definition 26, edges are not filtered by θ predicate. Furthermore, the resulting graph is stored in a bulk graph (Figure 4.6d on the preceding page) where only the vertices id from the two graph operators appear as pairs. This last operation takes time $k_1 + k_2 + \sum_{h \in HI} \left(b_1^h \cdot b_2^h + out_1^h \cdot out_2^h \right)$ where b_i^h is the size of the h bucket for the i-th operand, while out_i^h is the outgoing vertices' size for all the vertices within the h bucket for the i-th operand.

Such algorithm could be also extended to the disjunctive semantics as outlined in Algorithm II.2, where `isDisjunctive` activates the part concerning the evaluation of the disjunctive semantics. All the edges discarded from the intersection in line 26 for $u \oplus u'$

Algorithm II.2 Graph Disjunctive EquiJoin Algorithm: Join Phase

```
1:  procedure DisjunctiveEdges($G_1, G_2, \langle HI, \hbar, \Delta E_1, \Delta E_2, f \rangle$):
2:      $offsets$ :=find($\hbar$, $HI$)
3:      if $\hbar$.offset$_1$ ≠NULL and $offsets$.offset$_2$ ≠NULL then
4:          ▷ Checking all the left graph edges that have a target vertex which will be returned...
5:          for each $e \in \Delta E_1, v' \in VertexVals_2[offsets.\text{offset}_2]$ do
6:              if $\theta'(e.\text{outv}, v')$ then
7:                  $f$.Write(E={$(u \oplus u') \xrightarrow{e} (e.\text{outv} \oplus v')$})
8:      else if $\hbar$.offset$_2$ ≠NULL and $offsets$.offset$_1$ ≠NULL then
9:          ▷ ...and the dual case for the right graph operand
10:         for each $v \in VertexVals_1[offsets.\text{offset}_1], e' \in \Delta E_2$ do
11:             if $\theta'(v, e'.\text{outv})$ then
12:                 $f$.Write(E={$(u \oplus u') \xrightarrow{e'} (v, e'.\text{outv})$})
13:
14: procedure DisjunctivePartitionHashJoin($G_1, G_2, \theta$, isDisjunctive = true):
15:     $\theta'(u,u') := \theta(u,v) \wedge (u \oplus u')(A_v) = u \wedge (u \oplus u')(A'_v) = u'$;
16:     $\Theta'(e,e') := (e \oplus e')(A_e) = e \wedge (e \oplus e')(A'_e) = e'$
17:     $HI$ = IntersectHashes($HashOffset_1$,$HashOffset_2$).iterator();
18:     File $AdjFile$ = Open(), $DisjFile$ = Open();
19:     while $HI$.hasNext() do
20:         $h$ = $HI$.next();
21:         for each $u \in VertexVals_1[h.\text{offset}_1], u' \in VertexVals_2[h.\text{offset}_2]$ do
22:             if $\theta'(u,u')$ then
23:                 $AdjFile$.Write(V={$u \oplus u'$},)
24:                 for $h_{1,2}$ ∈UnionHashes($out_V(u)$,$out_{V'}(u')$).iterable() do
25:                     if $h_{1,2}$.offset$_1$ ≠NULL and $h_{1,2}$.offset$_2$ ≠NULL then
26:                         ▷ Conjunctive case: we have a match between the outgoing elements
27:                         BitMap $LE$ := new BitMap($out_V(u)[h_{1,2}.\text{offset}_1]$);
28:                         BitMap $RE$ := new BitMap($out_{V'}(u')[h_{1,2}.\text{offset}_2]$);
29:                         for each edge $e \in out_V(u)[h_{1,2}.\text{offset}_1], e' \in out_{V'}(u')[h_{1,2}.\text{offset}_2]$ do
30:                             if $\theta'(e.\text{outv}, e'.\text{outv})$ and $\Theta'(e,e')$ then
31:                                 $AdjFile$.Write(E={$e \oplus e'$}); $LE$.remove($e$); $RE$.remove($e'$);
32:                         ▷ Checking which of the unmatched edges satsfy the disjunctive semantics
33:                         if isDisjunctive then DisjunctiveEdges($G_1, G_2, \langle HI, h_{1,2}, LE, RE, DistFile \rangle$);
34:                     else if isDisjunctive then
35:                         DisjunctiveEdges($G_1, G_2, \langle HI, h_{1,2}, out_V(u)[h_{1,2}.\text{offset}_1], out_{V'}(u')[h_{1,2}.\text{offset}_2], DistFile \rangle$);
```

(cf. line 31 in Algorithm II.1) should be considered (line 35), either if they come from the left operand (line 4) or from the right one (line 9). Among all such edges, we can consider first the ones coming from the first graph operand: since the final edges must only connect vertices belonging to the final vertex set, we consider only those e that have a destination vertex "e.outv" which hash value appears in HI (line 2). Moreover it has to satisfy the binary predicate θ (line 6) jointly with another vertex v', coming from the opposite operand (line 4). Hence we establish (e.g.) an edge $(u \oplus u', e.\text{outv} \oplus v')$ having the same values and attributes of e' and the same set of labels (line 7), and stored into a different file. Similar considerations should be done by the edges discarded from the conjunctive phase (line 33).

4.4.2 Experimental Evaluation

Through the following experiments we want to prove that *(i)* both hash buckets and memory mapping for the graph join operands provide better results for GCEA, *(ii)* which outperforms the query plans for other query languages (both graph and relational). For the first case we have to use graph libraries or graph databases where transactions and logging can be disabled, while for the second we choose state of the art graph databases implementing specific query languages.

In order to do so we choose the simplest graph representation that provides better performances for all the addressed languages: we choose a graph where only vertices contain values and where labels are stored in both vertices and edges. We created our data using the LiveJournal Graph [LLDM09] containing 4,847,571 unlabelled vertices and 68,993,773 unlabelled edges. Each vertex represents a user which is connected to each of its friends by an edge. Since no data values are given within the datasets, we enriched the graph using the guidelines of the LDBC Social Network Benchmark protocol [EALP+15], and hence associated to each user an IP address, an Organization and the year of employment[3]. For each experiment, the input data were obtained by starting a random walk from the same vertex but using a different seed for the graph traversal. New data sets were obtained incrementally by visiting each time a number of vertices that is a power of 10, from 10 to 10^6.

We performed our tests over a MacOsX with a 2.2 GHz Intel Core i7 processor and 16 GB of RAM at 1600 MHz, and an SSD Secondary Storage with an HFS file system. We evaluate the graph join using as operands two distinct sampled subgraphs with the same vertex size ($|V|$), where the θ predicate is the following one: $\theta(u,v) \overset{def}{=} u.Year1 = v.Year2 \wedge u.Organization1 = v.Organization2$. Such predicate does not perform a perfect 1-to-1 match with the graph vertices, thus allowing to test the algorithm with different multiplicities values. We tested the algorithm with the conjunctive semantics, having a subset of the operations of the disjunctive one.

4.4.2.1 Evaluating Data Structures

We benchmark our solution with graph data models where database transactions either do not exist or can be disabled. We first consider two graph libraries accessing graphs in main memory; we tested the Boost Graph Library 1.60.0 with the most efficient configuration for graph traversals tasks, *vec* [SLL02], and Snap 3.0 [LS16] considering the

[3] The resulting enriched graph is available at http://rebrand.ly/c2e0e. The repository at http://rebrand.ly/unibo1516a provides our full source code including 1) our graph model implementation in both Java and C++, plus the queries in SPARQL and Cypher.

Operands Size		**GCEA running time, result creation excluded**				**GCEA result creation time**			
Left ($\|V\|$)	Right ($\|V\|$)	Proposed	Boost	SNAP	Sparksee	Proposed	Boost	SNAP	Sparksee
10	10	0.19 ms	**0.09**×	0.23×	9.42×	**0.0010** ms	17.00×	36.40×	738.33×
100	100	0.18 ms	**0.85**×	1.72×	24.96×	**0.0023** ms	5.39×	17.04×	290.14×
1 000	1 000	**0.31** ms	5.68×	14.93×	88.42×	**0.0036** ms	7.72×	14.67×	215.65×
10 000	10 000	**1.90** ms	11.13×	26.83×	156.42×	**0.3706** ms	4.60×	7.61×	15.67×
100 000	100 000	**32.31** ms	8.73×	19.33×	81.05×	**39.3428** ms	4.20×	5.80×	11.70×
1 000 000	1 000 000	**332.60** ms	15.42×	33.15×	171.54×	**3,207.8738** ms	5.76×	12.29×	15.50×

(a) This table shows a comparison between different data structures while performing the GCEA algorithm, and when the operands are already loaded: Boost and SNAP load the operands in primary memory, Sparksee load only some indices, while our data structure leaves everythin in secondary memory. The first part of the table shows that the indexing structure becomes relevant for big data, when our data structure starts to outperform the most efficient data structure, Boost. The second part shows that our data structure allows a fast serialization of the resulting adjacency graph.

Left ($\|V\|$)	Right ($\|V\|$)	Proposed	Boost	SNAP	Sparksee
10	10	0.23 ms	**0.68**×	0.98×	7.73×
100	100	**0.50** ms	1.60×	5.22×	11.76×
1 000	1 000	**3.38** ms	1.68×	6.94×	13.47×
10 000	10 000	**34.26** ms	1.52×	7.25×	13.84×
100 000	100 000	**355.96** ms	1.47×	6.27×	14.73×
1 000 000	1 000 000	**3,518.47** ms	1.89×	6.10×	17.79×

(b) Graph operand creation+storing time. This table represents the cost of SerializeOperand where each operand is stored in the data representation of choice. Given that the other data representation do not allow to index the graphs, the indexing step is performed only over our proposed data structure.

Table 4.1 Benchmarking results for the LiveJournal database over C++ graph libraries and low level databases.

Operands Size		**Result**		**Join Time (C/C++)** (ms)			**Join Time (Java)** (ms)	
Left ($\|V\|$)	Right ($\|V\|$)	Size ($\|V\|$)	Size ($\|E\|$)	Virtuoso	PostgreSQL	GCEA (C++)	Neo4J	GCEA (Java)
10	10	5	2	4.99	11.29	0.53	211.45	24.97
10^2	10^2	16	4	4.94	22.82	0.93	222.87	32.70
10^3	10^3	251	55	4.55	22.92	4.35	448.97	117.58
10^4	10^4	2,734	680	117,712.00	183.90	40.42	3,149.90	1,150.37
10^5	10^5	26,803	7,368	>4H	7,150.74	411.78	241,026.79	17,178.49
10^6	10^6	151,212	99,558	>4H	99,683.91	3,966.72	>4H	178,066.80

Table 4.2 Graph Join Running Time. Each data management system is grouped by its graph query language implementation. This table clearly shows that the definition of our query plan clearly outperforms the default query plan implemented over those different graph query languages and databases.

attributes only over the vertices (`TNodeNet<TAttr>`). Then we consider the Sparksee* graph database [DSUBGVn+10]: transactions were disabled in the configuration file, as well as logging, rollback and recovery facilities. Concerning the graph database management implementation, no assumptions can be made as it is closed source.

We implemented our graph join algorithm for all the aforementioned libraries. We used the standard graph library methods to store the graph in secondary memory (serialization or graph database storage) and extended the PartitionHashJoin by doing a preliminary vertex bucketing phase: buckets are not supported and vertices cannot be sorted by hash value.

Join Evaluation Time

In this case we evaluate two aspects: *(i)* the join algorithm running time and *(ii)* the time required to create the solution and store it in secondary memory.

Table 4.1a provides the cost of performing the sole join algorithm excluding the result storing time. All the competitors' graphs were joined through GCEA and vertices with the same hash were put in the same bucket in main memory. It must be emphasised that both Boost and SNAP operands were loaded in primary memory, while our operands were accessed in secondary memory through memory mapping. The table shows how all the other data structures had a worse performance due to the initial cost of the bucket creation and sorting. We must also remark that this result justifies the need of our data structure for the proposed algorithm. The same table provides the time required to store the results as an adjacent list in secondary memory using the default graph library representation (non-labelled vertices and edges, default serialization). In this case our solution always outperforms the other graph libraries and databases.

Operand creation time

We consider the graph creation time in main memory and the cost of storing it in secondary memory per operand (Table 4.1b). For both Boost and SNAP the default serialization methods are performed, while for Sparksee* we simply closed the database. In this case our solution outperforms all the competitors.

4.4.2.2 Join Execution Time

This last experiment compares the interpretation of query plans for both relational and graph databases with GCEA.

It is necessary to compare the performances of our algorithm with query languages running on top of property graph databases because our physical model generalizes property graphs. Among the Property Graph databases we do not consider SQLGraph [SFS+15] because there is no existing implementation and, most importantly, the Gremlin query language allows only to perform graph traversal queries returning bag of values.

We used default configurations for both Neo4J and PostgreSQL, while we changed the cache buffers configurations for Virtuoso (as suggested in the configuration file) for 16GB of RAM. We kept the default multithreaded query execution plan.

We choose to perform our tests over **Neo4J** using **Cypher** as a query language, because Neo4J allows to extend the built-in query plans with ad hoc solutions [HG16], eventually allowing an implementation of our algorithm in a future. Cypher queries were sent using the Java API but the graph join operation was performed only in Cypher through the `execute` method of an `GraphDatabaseService` object. Neo4J graphs were fine tuned by indexing the attributes `Organization` and `Year` involved in the query and, since Cypher language does not allow to access to different graphs, both graph join operands were stored within the same graph.

PostgreSQL queries were evaluated directly through the `psql` client and benchmarked using both `explain analyse` and `\timing` commands.

Virtuoso was benchmarked through iODBC connection evoked in C using Redland RDF library: no HTTP connections were used and only the `librdf_model_query_execute` function was involved in the graph join operation. Virtuoso prefer to index triplets per patterns and do not allow triplet indexing by values. This allows us to query each property graph with SPARQL query language, specifically targeted for triplestores, through their RDF representation. Indexing structures were not tuned, as a default set of indices are defined during the graph creation, and data is automatically indexed. we also took into account that both input and output met the requirements of Definition 15 on page 74.

All the aforementioned conditions do not degrade the query evaluations.

Table 4.2 represents the result of such benchmarks. The competitors' join time is made up only by the query evaluation time, while our proposed implementation considered the whole GCEA algorithm, and hence both the partitioning phase, the operands' serialization and the actual join execution were considered. As a result our solutions always outperform the competitors' query plans within their own language implementation.

Such performances quickly degrade due to both the sparsity of the data representation requiring to perform more *path joins* than the ones required for the property graph model. **Cypher** uses a pipe query evaluation model allowing to refine queries in further steps. Regarding the implementation of the graph conjunctive join operator in Cypher, *ValueHashJoins* are performed between vertices coming from different graph operands, and hash values are either evaluated at run time, or depend on attributes' values indexings. This choice supports the experimental evidence of Cypher having a better scalability than SPARQL, where RDF graphs cannot be indexed by values (see the next paragraph). Once the Cypher query is transformed into a pipe-based query plan, most of the pipes' sources appear to be *NodeByLabelScan* and *AllNodeScan*: this means that all the graph's vertices (with a given label) are considered in the first steps of computation. As a result the query plan scans more data than it should to provide the final result. In our algorithm this drawback does not occur because we directly access the data per buckets on both graph operands, avoiding to consider any vertices' combinations that will not appear in the final result.

4.5 Graph Less-Equal Join

In this section we're going to extend the previous algorithm to support less equal predicates, e.g. $A_i \leq A_j$. Hereby, in this scenario the `generateHash` function must associate A_i values for the left operand and the A_j values for the second operand. Given that our proposed GCEA algorithm is based on an extension of the sort merge join relational algorithm, we can simply extend it to support less-equal joins. Hereby, the only part of Algorithm II.1 that has to be rewritten is the PartitionHashJoin procedure. The parts in which such algorithm differs from Algorithm II.3 are remarked with blue text or background colour: in particular, instead of selecting the hash values that are in common between the two vertices' sets, we extract hashes h_1 for the first graph and h_2 for the second such that $h_1 \leq h_2$. For our experimental evaluation, we choose A_i and A_j to be the years of employment (`Year1` and `Year2`).

Figures 4.7 and 4.8 respectively represent the less-equal join for Cypher and SPARQL, that is going to be benchmarked against our algorithm. If we compare such queries to the ones firstly presented for the equi-join query (respectively Figure 4.3 on page 93 and 4.4) we can see that the only difference was the replacement of the equivalence predicate with the less equal one. The results of such queries evaluations are provided in Table 4.3: we used the same experimental environment and dataset described in Section 4.4.2 on page 105 for the previous algorithm. The relational database is always more performant

```
MATCH (src1)-[:r]->(dst1),
     (src2)-[:r]->(dst2)
WHERE src1.Year <=src2.Year2 AND dst1.Year <=dst2.Year2 AND src1.graph='L'
    ↪  AND src2.graph='R' AND dst1.graph='L' AND dst2.graph='R'
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 ,  MyGraphLabel
    ↪ :"U-"})-[:r]->(:U {Organization1:dst1.Organization1, Organization2
    ↪ :dst2.Organization2 , Year1:dst1.Year1, Year2:dst2.Year2,
    ↪ MyGraphLabel:"U-"}) return p
UNION ALL
MATCH (src1)-[:r]->(u), (src2)-[:r]->(v)
WHERE  src1.Year <=src2.Year2 AND src1.graph='L' AND src2.graph='R' AND
    ↪ (( u.Year1>v.Year2))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :"U-"}) return p
UNION ALL
MATCH (src1)-[:r]->(u), (src2)
WHERE src1.Year <=src2.Year2 AND src1.graph='L' AND src2.graph='R' AND (
    ↪ NOT ((src2)-[:r]->()))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :"U-"}) return p
UNION ALL
MATCH (src1), (src2)-[:r]->(v)
WHERE src1.Year <=src2.Year2 AND dst1.Year <=dst2.Year2 AND src1.graph='L'
    ↪  AND src2.graph='R' AND (NOT ((src1)-[:r]->()))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :"U-"}) return p
UNION ALL
MATCH (src1), (src2)
WHERE src1.Year <=src2.Year2 AND src1.graph='L' AND src2.graph='R' AND (
    ↪ NOT ((src2)-[:r]->())) AND (NOT ((src1)-[:r]->()))
CREATE p=(:U {Organization1:src1.Organization1, Organization2:src2.
    ↪ Organization2 , Year1:src1.Year1, Year2:src2.Year2 , MyGraphLabel
    ↪ :"U-"}) return p
```

■ **Figure 4.7** Cypher implementation for the graph less-equal join operator. Please note that the equivalence predicate has changed to <=, while the pattern matching parts are fixed.

```
CONSTRUCT {
        ?newSrc <http://jackbergus.alwaysdata.net/graph> "Result";
              <http://jackbergus.alwaysdata.net/edges/result> ?newDst;
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip1;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org1;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y1;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip2;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org2;
              <http://jackbergus.alwaysdata.net/property/Year2> ?y2.
        ?newDst <http://jackbergus.alwaysdata.net/graph> "Result";
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip3;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org3;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y3;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip4;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org4;
              <http://jackbergus.alwaysdata.net/property/Year2> ?y4.
}
FROM NAMED <leftpath/to/graph>
FROM NAMED <rightpath/to/graph>
WHERE
{
  GRAPH ?g {
                ?src1 <http://jackbergus.alwaysdata.net/property/Id> ?id1;
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip1;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org1;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y1.
        }.
  GRAPH ?h {
                ?src2 <http://jackbergus.alwaysdata.net/property/Id> ?id2;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip2;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org2;
              <http://jackbergus.alwaysdata.net/property/Year2> ?y2.
        }
  filter(?g=<leftpath/to/graph> &&
         ?h=<rightpath/to/graph> &&
         xsd:integer(?y1) <= xsd:integer(?y2))

  BIND (URI(CONCAT("http://jackbergus.alwaysdata.net/values/",?id1,"-",?id2)) AS ?newSrc)

  OPTIONAL {
                GRAPH ?g {
                        ?src1 <http://jackbergus.alwaysdata.net/edges/edge> ?dst1.
                        ?dst1 <http://jackbergus.alwaysdata.net/property/Id> ?id3;
              <http://jackbergus.alwaysdata.net/property/Ip1> ?ip3;
              <http://jackbergus.alwaysdata.net/property/Organization1> ?org3;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y3.
                }.
                GRAPH ?h {
                        ?src2 <http://jackbergus.alwaysdata.net/edges/edge> ?dst2.
                        ?dst2 <http://jackbergus.alwaysdata.net/property/Id> ?id4;
              <http://jackbergus.alwaysdata.net/property/Ip2> ?ip4;
              <http://jackbergus.alwaysdata.net/property/Organization2> ?org4;
              <http://jackbergus.alwaysdata.net/property/Year1> ?y4.
                }
                FILTER ( xsd:integer(?y3) <= xsd:integer(?y4) )
                BIND (URI(CONCAT("http://jackbergus.alwaysdata.net/values/",?id3,"-",?id4))
                     ↪ AS ?newDst)
        }
}
```

Figure 4.8 SPARQL implementation for the graph less-equal join operator. Even in this case, the pattern matching is kept the same, while the only change is with the predicate <=.

Size (L,R)	Neo4J	PostgreSQL	Virtuoso	**GLEA C++** (ms)
10^1	$9,359.86\times$	$82.61\times$	$37.58\times$	0.15
10^2	$2,374.98\times$	$21.57\times$	$10,308.09\times$	1.17
10^3	$28,368.97\times$	$191.32\times$	$> 3.6\cdot 10^6$ ms	10.72
10^4	$> 3.6\cdot 10^6$ ms	$1616.22\times$	$> 3.6\cdot 10^6$ ms	98.50
10^5	$> 3.6\cdot 10^6$ ms	$> 3.6\cdot 10^6$ ms	$> 3.6\cdot 10^6$ ms	1,016.44
10^6	$> 3.6\cdot 10^6$ ms	$> 3.6\cdot 10^6$ ms	$> 3.6\cdot 10^6$ ms	12,583.89

■ **Table 4.3** Graph Less-Equal Join running time. GLEA considers the time required to write the solution into secondary memory. Each data management system is compared with the most efficient implementation of GLEA in C++. Please note that the graph creation time, which is an order of magnitude less (Table 4.1b) than the GLEA execution time.

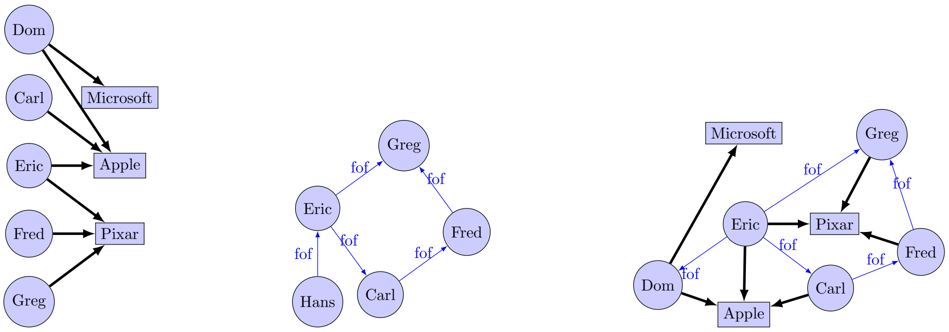
than the two graph databases, thus suggesting that the relational databases have more efficient query optimizations for less-equal predicates than graph databases. This final result remarks that the presented algorithm could be easily extended to fit other query evaluation scenarios.

Please note that the same algorithm can be also used when we have a conjunction between less-equal predicates, $A_i \leq A_j \wedge A_k \leq A_h$ and if there is a lexicographical order between the elements $A_i \times A_k$ and $A_j \times A_h$. In these cases we can generate a hashing function h such that if $i \leq j \wedge k \leq h$ holds, then the order is reflected by the hashing function $h(i,k) \leq h(j,h)$ [BBPV11].

Algorithm II.3 Graph Less-Equal join Algorithm (GLEA)

```
procedure LEqJoin($G, G', \theta$)
    $hashFunction$ = generateHash($\theta$);
    $omap_1$ = OperandPartitioning($G$,$hashFunction$)
    $omap_2$ = OperandPartitioning($G'$,$hashFunction$)
    $\overline{G}_1$ = SerializeOperand($G$,$omap_1$)
    $\overline{G}_2$ = SerializeOperand($G'$,$omap_2$)
      return PartitionHashJoin($\overline{G}_1, \overline{G}_2, \theta$)

procedure PartitionHashJoin($G_1, G_2, \theta$):
    $\theta'(u,u') := \theta(u,v) \wedge (u \oplus u')(A_v) = u \wedge (u \oplus u')(A'_v) = u'$;
    $\Theta'(e,e') := (e \oplus e')(A_e) = e \wedge (e \oplus e')(A'_e) = e'$
    for $h_1$ ∈MinHashIterator($HashOffset_1$) do
        for $h_2$ ∈MaxHashIterator($HashOffset_2$) s.t. $h_1 \leq h_2$ do
            for each $u \in VertexVals_1[h_1.\text{offset}_1]$, $u' \in VertexVals_2[h_2.\text{offset}_2]$ do
                if $\theta'(u,u')$ then
                    $AdjFile$.Write(V={$u \oplus u'$},)
                    for $h'_1$ ∈MinHashIterator( $out_V(u)$) do
                        for $h'_2$ ∈MaxHashIterator($out_{V'}(u')$) s.t. $h'_1 \leq h'_2$ do
                            for each edge $e \in out_V(u)[hparticular'_1.\text{offset}_1]$, $e' \in out_{V'}(u')[h'_2.\text{offset}_2]$ do
                                if $\theta'(e.\text{outvertex}, e'.\text{outvertex})$ and $\Theta'(e,e')$ then
                                    $AdjFile$.Write(E={$e \oplus e'$})
```



(a) Company Membership, Graph $\mathcal{A}$ **(b)** Social Network Friendship, Graph $\mathcal{N}$ **(c)** Merged Graph, $\mathcal{A} ⟕^{\vee}_{\sim} \mathcal{N}$

4.6 Left, right and full graph joins.

The following example motivates the need of an extension of the aforementioned graph operators, since they cannot describe all the possible join combinations for graphs.

▸ **Example 19.** *Suppose to have a company membership bipartite graph $\mathcal{A}$ (Figure 4.9a), where each employee is associated to a company where (s)he works. Suppose now to have another graph $\mathcal{N}$ (Figure 4.9b) providing a social network where some users are provided. Given that we can infer a predicate $\sim$ which finds correspondences between each employee in $\mathcal{A}$ and their on-line account in $\mathcal{N}$, join the graphs so that $\mathcal{A}$ is enriched with the friendship edges from $\mathcal{N}$ as in Figure 4.9c. For this toy example, such $\sim$ function associates the employee and the social network users that have the same name.*

Concerning the edges, we must adopt a disjunctive semantics, because the edges in $\mathcal{A}$ link employees to companies, while the ones in $\mathcal{N}$ link friends among them. As a consequence, such edges could never match with a conjunctive semantics: in order to preserve the `worksFor` *relationships in $\mathcal{A}$ and inherit the friendship relationships* `fof` *among employees, we must adopt the disjunctive semantics.*

Concerning the vertices that have to be returned by the graph join operation, we can observe that graph $\mathcal{N}$ contains more users than the employees in $\mathcal{A}$, and that $\mathcal{A}$ contains employees that have no account in $\mathcal{N}$. Since is $\mathcal{A}$ the graph to be enriched, we want to preserve all the information in $\mathcal{A}$ and to discard all the users in $\mathcal{N}$. We can even observe that the companies are not included in the definition of $\sim$, and hence a $V_{\mathcal{A}} \bowtie_{\sim} V_{\mathcal{N}}$ between the vertices shall not return any company. For this reason, we have to perform the left join $V_{\mathcal{A}} ⟕_{\sim} V_{\mathcal{N}}$ over the vertices, so that all the employees and the companies from graph $\mathcal{A}$ are preserved, while the users in $\mathcal{N}$ that are not employees are not represented in the final graph.

We must observe that the operation outlined by the previous example is not matched by the former definitions of graph joins. Consequently, we now introduce the outer join operators by extending the join's vertex definition: if we want to include only the unmatched vertices belonging to the left join operand then we have a **left join** (⟕), which is defined as follows:

▸ **Definition 27** (Graph Left θ-Join)**.** *Given two graphs $G_a = (V, E, A_v, A_e)$ $G_b = (V', E', A'_v, A'_e)$, a **graph left θ-join** is defined by extension of Definition 26 as follows:*

$$G_a ⟕^{\mathbf{es}}_{\theta} G_b = (V ⟕_{\theta} V', E_{\mathbf{es}}, A_v \cup A'_v, A_e \cup A'_e)$$

where $\mathbin{⟕}_\theta$ the left outer θ-join among the vertices. ◂

If we symmetrically include only the vertices from the right graph then we have a **right join** ($\rtimes$), and thus the following definition can be provided:

▸ **Definition 28** (Graph Right θ-Join)**.** *Given two graphs $G_a = (V, E, A_v, A_e)$ and $G_b = (V', E', A'_v, A'_e)$, a **graph right θ-join** is defined as:*

$$G_a \rtimes_\theta^{\mathbf{es}} G_b = (V \rtimes_\theta V', E_{\mathbf{es}}, A_v \cup A'_v, A_e \cup A'_e)$$

where $\rtimes_\theta$ the right outer θ-join among the vertices. ◂

Given that both the conjunctive and the disjunctive semantics are expressible as joins among the edges and given the well known properties for the relational algebra operators, we have that for **es**$= \wedge$ or **es**$= \vee$ the following rewriting rule follows:

$$G_a \rtimes_\theta^{\mathbf{es}} G_b = G_b \mathbin{⟕}_\theta^{\mathbf{es}} G_a$$

If we want also to include the excluded vertices from both graphs we have a **full join** ($\mathbin{⟗}$). Similarly to the graph right θ-join, we can define this new operator either as a graph join where the vertices undergo a full θ-join or as a composition of the left and right join as follows when **es** is one of the two aforementioned edge semantics:

$$G_a \mathbin{⟗}_\theta^{\mathbf{es}} G_b = G_a \rtimes_\theta^{\mathbf{es}} G_b \cup G_a \mathbin{⟕}_\theta^{\mathbf{es}} G_b$$

As a consequence, we can say that the graph joins are a class of join operators, where an arbitrary combination $\otimes$ of vertices and **es** of edges is provided. Consequently, we can provide the following definition including all the previously defined graph join operators.

▸ **Definition 29** (Graph $\otimes_\theta$ product)**.** *Given two graphs $G_a = (V, E, A_v, A_e)$ $G_b = (V', E', A'_v, A'_e)$, the class of all the possible graph join operators is defined by the following **graph $\otimes_\theta$ product**:*

$$G_a \otimes_\theta^{\mathbf{es}} G_b = (V \otimes_\theta V', E_{\mathbf{es}}, A_v \cup A'_v, A_e \cup A'_e)$$

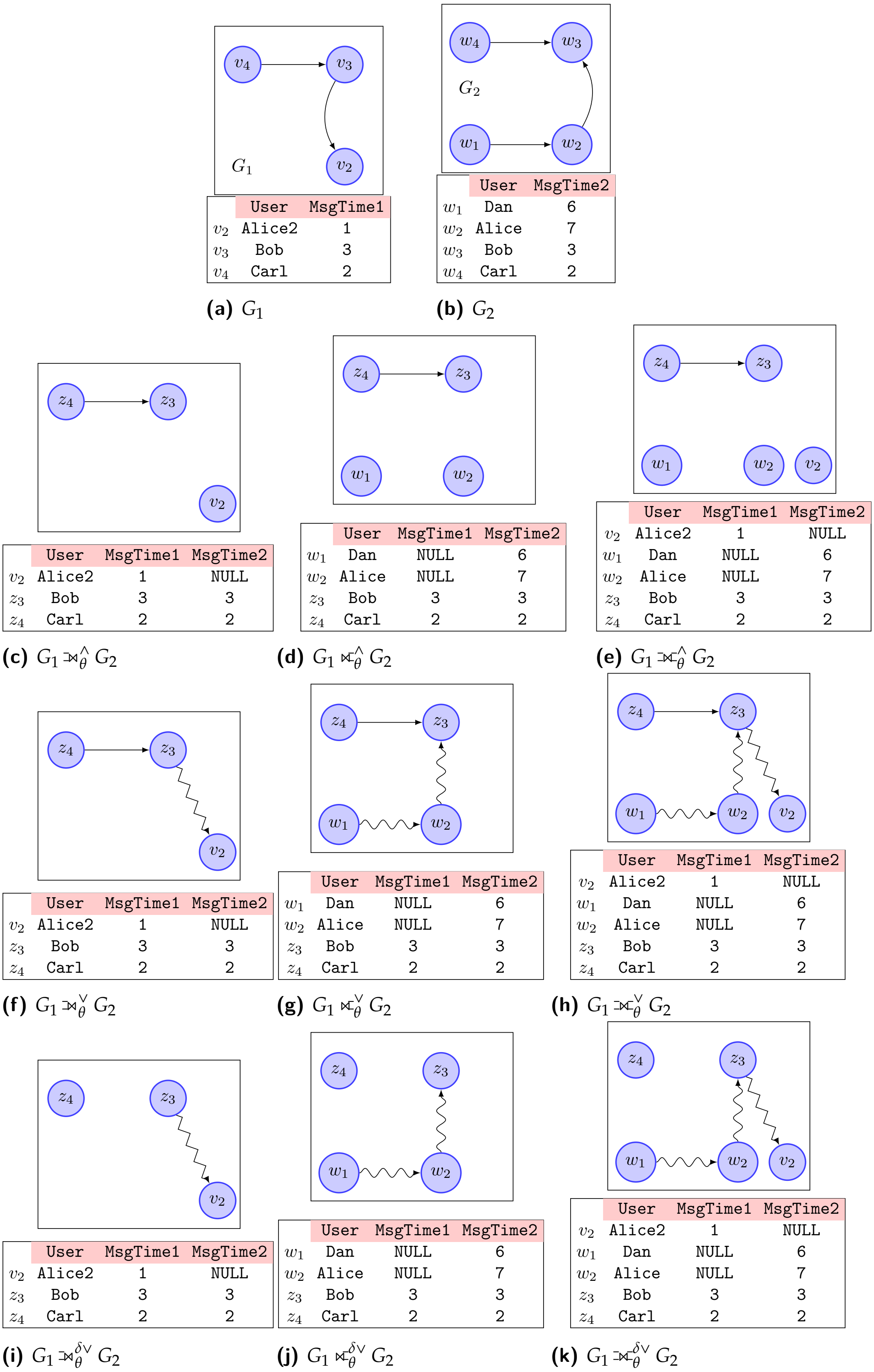
where $\otimes_\theta$ is an arbitrary θ-join operation among the vertices. ◂

▸ **Example 20.** *As a further example, let us take a look at Figure 4.10: suppose to have two graphs G_1 (a) and G_2 (b), describing interaction graphs between some users within different social network: any edge $u \overset{\delta}{\to} v$ describes that a user u.*`User` *sent a message to v.*`User` *at a time u.*`MsgTime`$_i$*, that received it and read it at time v.*`MsgTime`$_i$*. Given a predicate θ allowing to match the same users within different networks, we want to show which is the meaning of the different graph joins on top of these data structures.*

Generally speaking, the graph conjunctive semantics is used to retrieve the interactions occurring in different networks between the same users, even if at different times. The usage of the left (c), right (d) or full join (e) could be used to only change the result set of the vertices. Moreover, on all the three different cases of graph joins, the conjunctive semantics does not change the edges that are returned in the final result, since that semantics imply that the edges must appear in both graphs among the matched vertices.

As a consequence, the outer joins over the disjunctive semantics allow to return also the interactions coming either from the left operand (f), or from the right one (g) or both (h).

As a last step, we would like to intercept the interaction that only appear either in the left graph, or in the right one, or the interaction occurring either in the left graph or in the right one but not in



	User	MsgTime1
v_2	Alice2	1
v_3	Bob	3
v_4	Carl	2

(a) G_1

	User	MsgTime2
w_1	Dan	6
w_2	Alice	7
w_3	Bob	3
w_4	Carl	2

(b) G_2

	User	MsgTime1	MsgTime2
v_2	Alice2	1	NULL
z_3	Bob	3	3
z_4	Carl	2	2

(c) $G_1 ⟕^{\wedge}_{\theta} G_2$

	User	MsgTime1	MsgTime2
w_1	Dan	NULL	6
w_2	Alice	NULL	7
z_3	Bob	3	3
z_4	Carl	2	2

(d) $G_1 ⟖^{\wedge}_{\theta} G_2$

	User	MsgTime1	MsgTime2
v_2	Alice2	1	NULL
w_1	Dan	NULL	6
w_2	Alice	NULL	7
z_3	Bob	3	3
z_4	Carl	2	2

(e) $G_1 ⟗^{\wedge}_{\theta} G_2$

	User	MsgTime1	MsgTime2
v_2	Alice2	1	NULL
z_3	Bob	3	3
z_4	Carl	2	2

(f) $G_1 ⟕^{\vee}_{\theta} G_2$

	User	MsgTime1	MsgTime2
w_1	Dan	NULL	6
w_2	Alice	NULL	7
z_3	Bob	3	3
z_4	Carl	2	2

(g) $G_1 ⟖^{\vee}_{\theta} G_2$

	User	MsgTime1	MsgTime2
v_2	Alice2	1	NULL
w_1	Dan	NULL	6
w_2	Alice	NULL	7
z_3	Bob	3	3
z_4	Carl	2	2

(h) $G_1 ⟗^{\vee}_{\theta} G_2$

	User	MsgTime1	MsgTime2
v_2	Alice2	1	NULL
z_3	Bob	3	3
z_4	Carl	2	2

(i) $G_1 ⟕^{\delta\vee}_{\theta} G_2$

	User	MsgTime1	MsgTime2
w_1	Dan	NULL	6
w_2	Alice	NULL	7
z_3	Bob	3	3
z_4	Carl	2	2

(j) $G_1 ⟖^{\delta\vee}_{\theta} G_2$

	User	MsgTime1	MsgTime2
v_2	Alice2	1	NULL
w_1	Dan	NULL	6
w_2	Alice	NULL	7
z_3	Bob	3	3
z_4	Carl	2	2

(k) $G_1 ⟗^{\delta\vee}_{\theta} G_2$

Figure 4.10 Representation of outer joins over both conjunctive and disjunctive joins, where θ is defined as `MsgTime2≤MsgTime1`. Wiggled edges represent the edges obtained from right graph while zigzag edges represent the ones from left one. Straight-lined edges are the ones provided by the definition of the conjunctive join, and hence $E_{\bowtie}$. `NULL` values only appear in the tabular representation, while the provided vertex merge definition do not insert null values

both. In order to do so, we can define another edge semantics $\delta\vee$, removing from the disjunctive semantics the edges that appear in the conjunctive semantics, that is:

$$E_{\delta\vee} = E_{\vee} \backslash E_{\wedge}$$

After this step, we can finally obtain the graphs depicted in Figures 4.10i, 4.10j and 4.10k.

As a final example, we want to show how graph full joins may help in the creation of a common schema, generated from the two graph sources' schemas, thus showing how graph joins can be used for at both the data (D) and model (or schema, M) level.

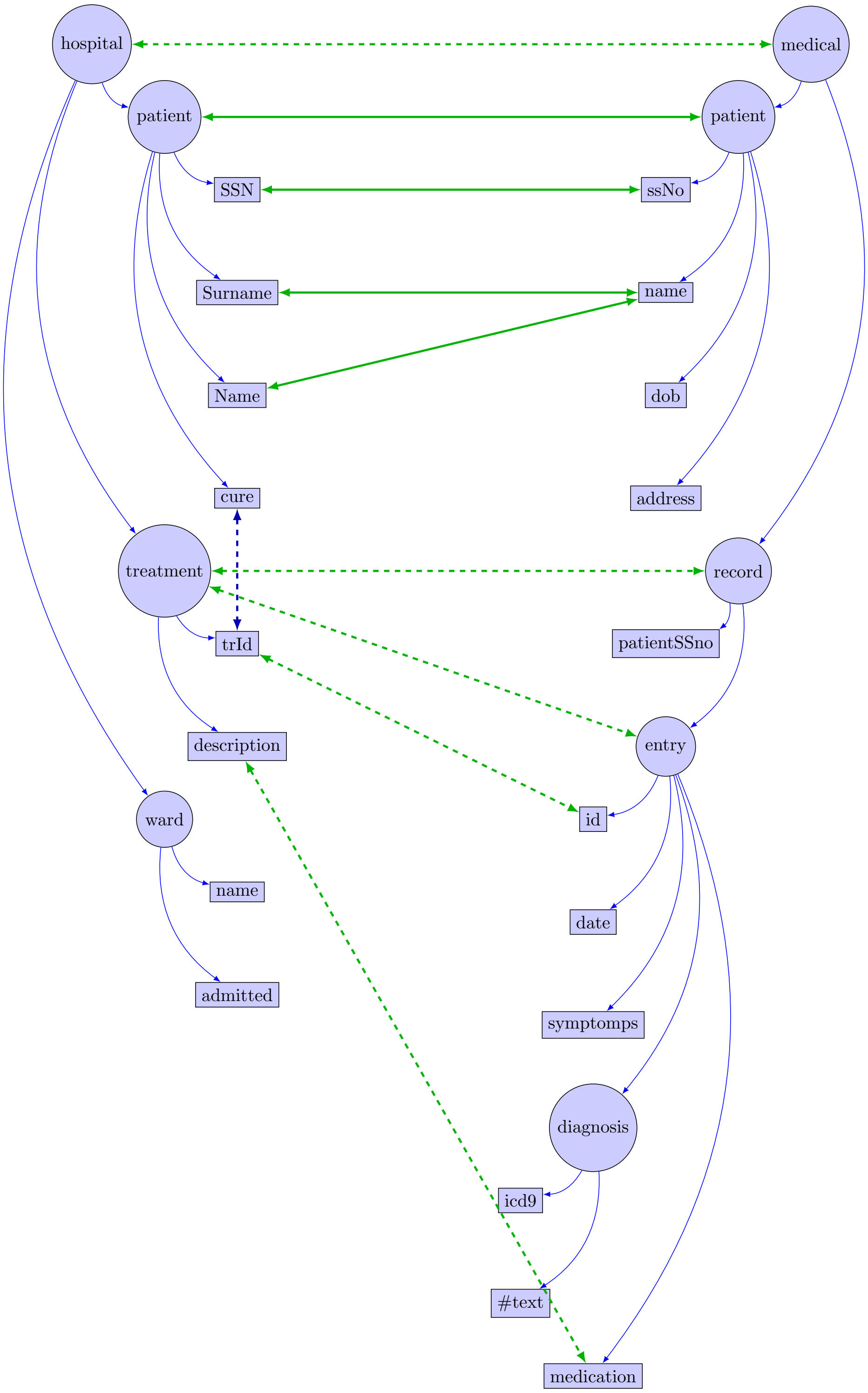
▸ **Example 21.** *Section 2.1.2 on page 21 introduced the operations of schema alignments between semistructured schemas. Given that literature expresses the containment relations through edges, Figure 4.11 on the next page represents the same structural representation of the former examples in a JSON schema format: now, each vertex represents an empty tuple, where only one single label is associated and therefore showed.*

Figure 4.12 provides the intermediate result of the schema integration previously provided in Figure 2.4b on page 23, where multiple matches are not resolved into one single entity (compare the previous `treatment_entry` *node with the current* `treatment-record` *and* `treatment-entry`*), where the two schemas where merged using a full join with a disjunctive semantics. This solution allows to preserve the non-matched information, while aggregates together the parts which have been matched within the alignment phase.*

4.7 Conclusions

This chapter showed the dualism between *logical model* used to represent data in a formal model and the *physical model* used for the final algorithms. While the former representation separates the operations on the vertices from the one over the edges, that are run in a subsequent step, the latter allows an implementation of the join algorithm over an adjacency lists combining both vertices and edges within the same step. The inefficiency of the first approach is also showed by the current graph data structures and graph databases, that provide a non scalable implementation of such join algorithms. We can also note that the bucketing phase allows to pre-process the graph in order to enhance the parallelization of the graph join algorithm. Furthermore, given the graph operator's commutative and associative properties, we plan to perform further studies for distributed graph multi-joins in order to check whether current relational query model proposed in [AGK+17] can be also used for graph data.

Last, it was showed that the graph join operators can be used both to combine data and schema representations; therefore, the graph join acts as a relevant graph operator for data integration at two distinct abstraction levels. Further investigations must be carried out in order to analyse whether it is useful to join graph data with their respective schema.



■ **Figure 4.11** Representing the containments in Figure 2.4a on page 23 as the ontology graphs presented in [ES13]: the left and the right graphs represent different schemas, where their blue edges express the containment relations. Green edges express either directly the result of the ontology alignment operation (continuous lines) or their refinements (dashed lines) that are going to be used to provide the integrated schema as a θ predicate.

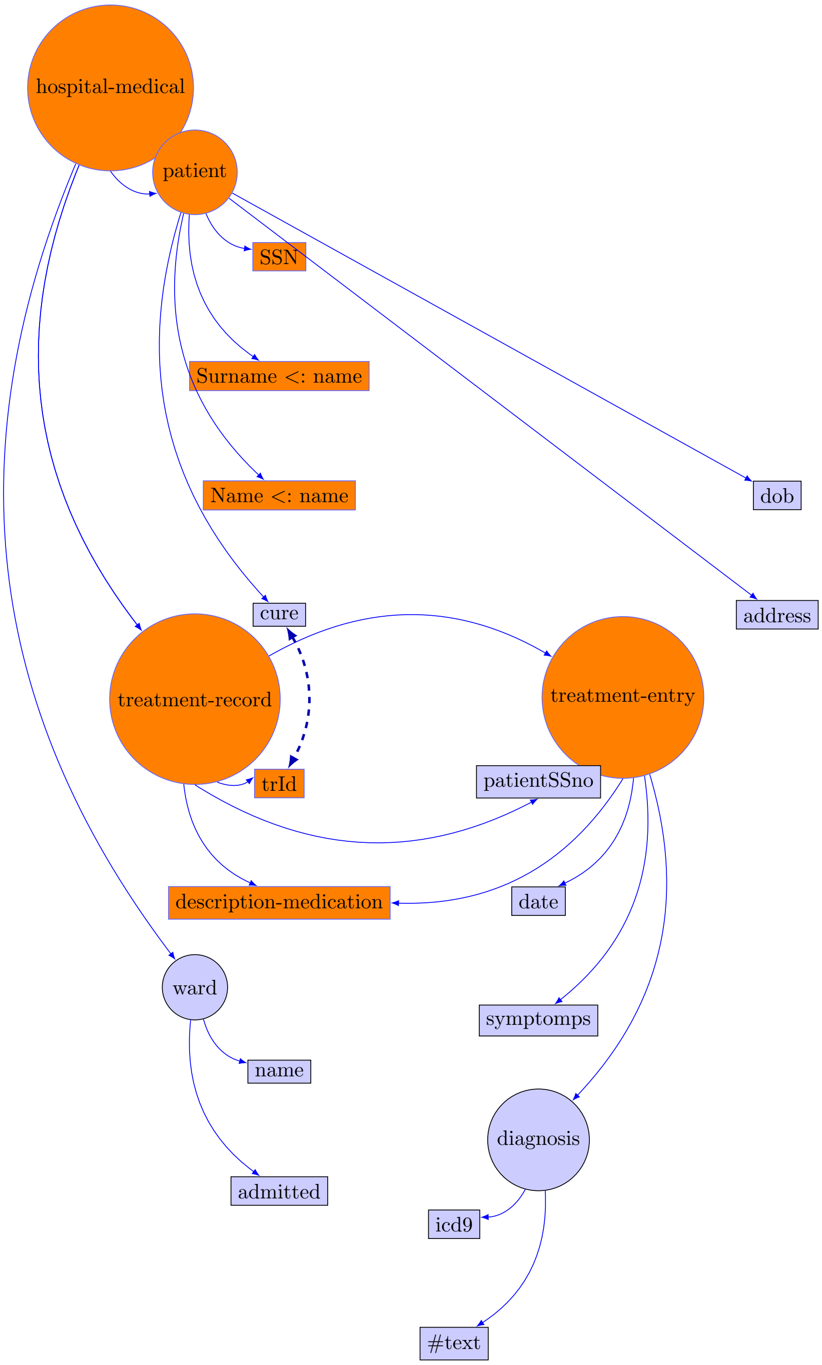


Figure 4.12 Representing an intermediate step for merging the two schemas via providing a graph full join with disjunctive semantics using the green edges as θ predicates. The vertices showed in orange represent the vertices that have been matched, and hence fused into one single vertex.

5 General Semistructured Model and Nested Graphs

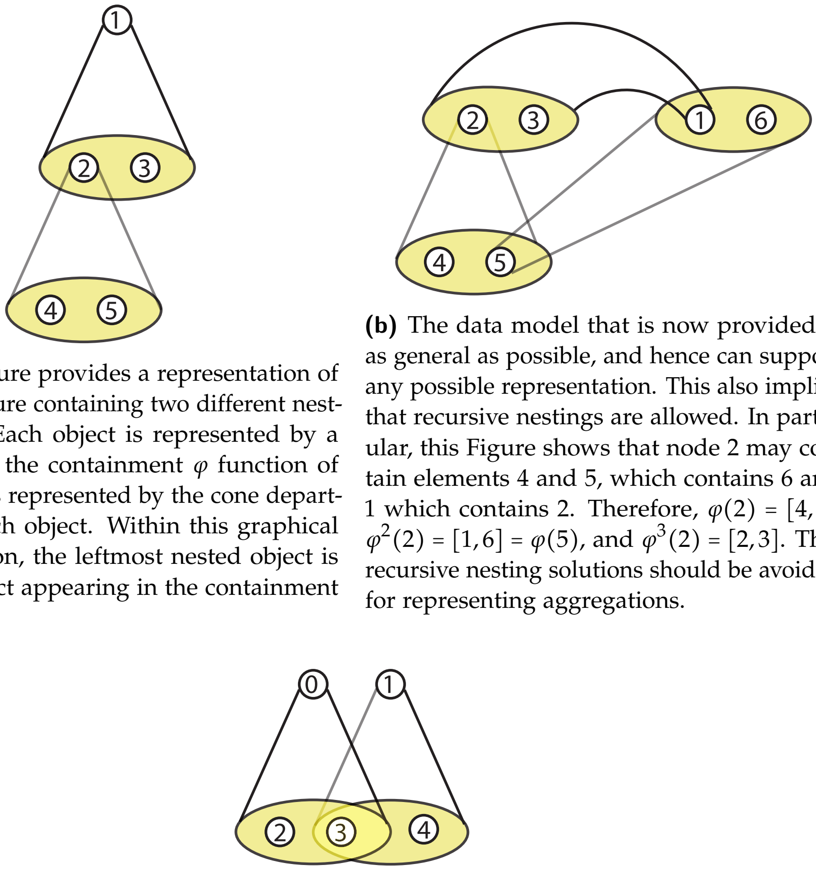
Contents

> *"The empirical basis of objective science has nothing 'absolute' about it. Science does not rest upon solid bedrock. The bold structure of its theories rises, as it were, above a swamp. It is like a building erected on piles. The piles are driven down from above into the swamp, but not down to any natural or 'given' base; and if we stop driving the piles deeper, it is not because we have reached firm ground. We simply stop when we are satisfied that the piles are firm enough to carry the structure, at least for the time being."*
>
> — Karl R. Popper, *The Logic of Scientific Discovery*, V. 30

Chapter 3 on page 55 addressed the problem of providing nested concepts in current data models. Graph models also proved not to meet such representation goals adequately and, therefore, a new data model for graph nested data is required. On the other hand, semistructured data representations do not allow to represent the containment of one single object by multiple containers, as needed for a possible generalization of the EPGM model. Consequently, we choose to define a Generalized Semistructured data Model (**GSM**, Section 5.1) on top of which the new nested graph model is going to be subsequently defined (**nested graphs**, Section 5.2).

Section 5.3 provides some translation functions τ from the previous data models towards the GSM data model, thus meeting two distinct goals: **(i)** we show the generality of GSM, that **(ii)** can be the used within the LAV/GAV data integration scenarios for syntactically τ-translating one data model towards the GSM or Nested Graph model. The choice of representing both vertices and edges with "objects" may surprise the reader, because Chapter 3 warned against the problem of semantic overloading. The reader must be aware that the implications of the previous paragraph fall only at the representation level of the single element while, on the other hand, the data model should distinguish those different concepts.

Last, we provide some use cases for both aggregating (Section 5.4.1) and integrating (Section 5.4.2) data representations within the proposed data format. Concerning the last use case, we must observe that in the third chapter we showed that the alignment of



(a) This picture provides a representation of a data structure containing two different nesting levels. Each object is represented by a circle, while the containment φ function of each object is represented by the cone departing from each object. Within this graphical representation, the leftmost nested object is the first object appearing in the containment function φ.

(b) The data model that is now provided is as general as possible, and hence can support any possible representation. This also implies that recursive nestings are allowed. In particular, this Figure shows that node 2 may contain elements 4 and 5, which contains 6 and 1 which contains 2. Therefore, $\varphi(2) = [4,5]$, $\varphi^2(2) = [1,6] = \varphi(5)$, and $\varphi^3(2) = [2,3]$. This recursive nesting solutions should be avoided for representing aggregations.

(c) This picture shows how the *GSM* model allows to nest one element (e.g. 3) in more than one single element. In particular, $\phi(0) \cap \phi(1) = [3]$.

Figure 5.1 Different possible instances of the *GSM* model, allowing arbitrary nestings within its objects. These pictures show some situations that cannot be natively expressed by the current nested relational model and other semistructured representations. In particular, each cone represents one single association between the containing object and its content through an expression p.

types – which may nest other concepts – relies on both the successive alignment of other attributes (or nested types) and on the attributes contained inside such types. Given that both types and attributes are involved in the alignment process, it follows that they all must be represented – within the alignment process – as objects with a given type represented through labels. After this alignment process, edges must be aligned too: this process is required for a complete data integration scenario. Consequently, this chapter introduces the concept of edge alignment for data integration (as previously introduced in Section 2.1.5 on page 29), thus allowing to provide a first definition of the ϱ operator as well as ς refining the previously mined alignments. Moreover, the same example shows that GSM allows a uniform representation between data models, schemas and alignments.

5.1 General Semistructured (Data) Model

After observing all the definitions provided in the previous sections and chapters, we shall now define a general framework allowing to include all the aforementioned nested representation, including the aggregation requirements provided in the previous sections. We must also consider that any object can nest several objects, and that any object can contain several others simultaneously.

Contrariwise to other approaches which distinguished atoms (e.g., numbers, strings) from objects (e.g., tuples, documents, object-oriented objects) and collections [MM06], the proposed Generalized Semistructured data Model provides a uniform representation of the three data representations. This characterisation implies the absence of a binding between a fixed schema and the data representation.

The uniform representation of objects and atoms permits collections supporting both representations uniformly. XML documents are an example where such situation already happens: in fact, both tags and text nodes may appear as the content of one single tag. This solution is achieved within our data model by representing both objects and values by their reference identifier o, and by using their identifier to insert them within such collections. The reference number provides both the associated expressions (list of atoms, $\xi(o)$) and labels (list of strings, $\ell(o)$). Moreover, we decide to represent atoms within the MetaModel, so that they can be later on used within transformation functions ς. Therefore, data can possibly embed both model and meta-model information.

However, all current data models (except from programming languages) do not allow a uniform representation of collections as objects and vice-versa. Given that objects may be represented as a property-value association, where values can be either atoms (as in the relational model and property graphs) or objects, we can assume that each property may be associated to a collection of object identifiers. In particular, our data model chooses to implement object o as multimap associations, which are expressed through a ϕ function, allowing to represent an object as a collection of (named) collections. In particular, each collection l represented within an object o can be referenced by using its identifying (and associated) property p, such that $\phi(o,p) = l$.

▸ **Definition 30** (General Semistructured (Data) Model)**.** *Given a* MetaModel *language MM expressing both constraints and values that can be checked by such constraints and a model M representable in MM ($\alpha(M) \subseteq MM$), any* ***object*** *o is directly represented by its object identifier $o \in \mathbb{N}$. Such object o shall belong to multiple model types $\ell(o) \in \wp(M)$, and has an associated list of expressions $\xi(o) \in \wp(MM)$ expressing either values or constraints associated to o. The attribute-value association embedded in each object o is expressed by the function ϕ: in particular $\phi(o,p) \in \wp(\mathbb{N})$ associates o and an attribute p (represented as an expression $p \in MM$) to a list of* *objects. $\varphi(o)$ provides all the objects contained by o for any p and is defined as follows:*

$$\varphi(o) \overset{\text{def}}{=} \bigcup_{p \in \mathcal{L}} \phi(o,p)$$

Therefore, any object o is identified by its id and is completely described by the functions ℓ, ξ and ϕ. Consequently, any instance of the GSM is described by the following quintuple:

$$GSM = (o, O, \ell, \xi, \phi)$$

where $O \subseteq \mathbb{N}$ is a finite set of object identifiers containing all the elements nested in the ***reference object*** *o at any nesting level ($\varphi^*(o) = O$), thus defining o as multiple collections of objects ($\varphi(o) \subseteq O$). Moreover, the labelling function ℓ, the expression and data function ξ and the containment function ϕ are both defined over o and over each element in O. Formally, we have $\ell : O \to \wp(M)$, $\xi : O \to \wp(MM)$ and $\phi : O \to MM \to \wp(O)$, which are all finite functions.*

◂

In particular, we are going to provide the *MM* language associated to GSMs in Section 5.1.1 on page 125. For the moment, we focus on the data representation aspects of GSM, and later on on the simple queries that can be performed on such model from the integration point of view (Section 5.4.1 on page 139).

▸ **Example 22.** *Figure 5.1a on page 120 provides an example of a GSM. In particular, this model can be modelled as follows:*

$$(1, \{1, 2, 3, 4, 5\}, \ell, \xi, \phi)$$

$$\phi(1,p) = [2,3] \qquad \phi(2,p) = [4,5]$$

The containment function maps the object to the contained objects by returning the last ones in a list. This data structure may become relevant in some domain specific applications, such as semistructured information, in which the order in which such data appear is relevant. Within our examples, we will graphically represent such containment order from left to right. Please also note that "o" can be arbitrarily chosen from any element in O.

Moreover, this data model represents objects as single atoms through ℓ and ξ, and as objects or collections. With respect to the current example, we can consider nodes 3, 4 and 5 with empty nestings as empty collections or objects.

At this point the following question arises: "*why identifiers* (also referred as "indices") *are crucial within nested data structures?*" As previously discussed, object identifiers also provide a solution to the data replication problem: by only storing the object identifier, we allow that one single piece of information is replicated in more that one single point within the database, thus allowing to overcome some limitations within semistructured data representations.

▸ **Example 22** (continuing from p. 122). *Figure 5.1c on page 120 represents the following GSM model:*

$$(0, \{0, 1, 2, 3, 4\}, \ell \xi, \phi)$$

$$\phi(0,p) = [2,3] \quad \phi(1,p) = [3,4]$$

In particular, we can see that element 3 is contained by both 0 and 1. This solution is only possible by using explicit identifiers that map objects to its content, typing and value information.

We can use this data model as the most general representation of data structures. Please note that this data model allows arbitrary nestings, such as "an element can contain itself" ($\exists o \in O.\exists p \in \mathcal{L}.o \in \phi(o,p)$) or "contain an element that, at any nesting level, contains the container object" ($\exists o,o' \in O.\exists n \in \mathbb{N}_{>0}.\exists p,p' \in \mathcal{L}.o' \in \phi(o,p) \wedge o \in \phi^n(o',p')$).

▸ **Example 22** (continuing from p. 122). *Figure 5.1b on page 120 provides an example of arbitrary nesting levels. In particular, if we choose to use* 2 *as the main element within our data model, we can have the following data model:*

$$(2,\{\,1,2,3,4,5,6\,\},\ell,\xi,\phi)$$

$$\phi(2,p) = [4,5] \quad \phi(5,p) = [1,6] \quad \phi(1,p) = [2,3]$$

If we want to focus on the arbitrary nesting levels of node 2*, we shall use the power notation over* φ*, where* φ^{n+1} *is recursively defined as the application of* φ *to any element of* φ^n*. Therefore, given that* $\varphi(4)$*,* $\varphi(6)$ *and* $\varphi(3)$ *map to the empty set, we can have the following situation:*

$$\varphi(2) = [4,5] \quad \varphi^2(2) = [1,6] \quad \varphi^3(2) = [2,3]$$

In particular, we are going to refer to any power n as ***(nesting) depth****. As we can see, this means that* 2 *may contain itself at the nesting depth of* 3*. Since that this model always allows arbitrary nesting levels, we will use the Kleene plus notation, which is defined as follows:*

$$\varphi^+(o) = \bigcup_{n \in \mathbb{N}_{>0}} \varphi^n(o)$$

Therefore, we can now express that 2 *contains itself at any level of nesting with the more compact notation* $2 \in \varphi^+(2)$*. Moreover, we can define* $\varphi^*(o)$ *as the union of* $\varphi^+(o)$ *with the singleton* $\{o\}$ *representing the missed application of* ϕ *over* o ($\forall x.\varphi^0(x) = \{x\}$).

We can observe that such model features violate the aggregation assumptions presented in Section 2.2.2 on page 40. The reader should remember that aggregations (that can be represented through ϕ containments) represent data abstractions: consequently, neither an object's abstraction can be represented by the containing object nor shall the containing object contain itself at any nesting depth. Since we want that our data model definition can be as slim and as general as possible, we want to express such restrictions as a side properties, similarly to the XML's DTD or RelaxNG. In particular, this thesis is going to focus on a subset of all the possible GSM-based data models (nested graphs included), which are the ones that avoid the recursive nestings presented in the previous example.

▸ **Axiom 1** (Nesting Loop Free). *A general semistructured data model GSM* $g \in (o,O,\ell,\xi,\phi)$ *is said to be nesting loop free if each object* $o \in O$ *does not contain itself at any nesting level. More formally:*

$$\forall o' \in O \cup \{\,o\,\}\,.\,o' \notin \varphi^+(o')$$

Objects $o \in O$ *having empty* ϕ *containments are referred as* ***leaves****.*

Moreover, we shall guarantee that the data operations on nesting loop free *GSM*s shall always return nesting loop free *GSM* results. This axiom also allows to define the hight at which one object is contained by the other without divergences or the possibility of having many possible nesting levels for the same object. Consequently, we can define a "relative height" definition, where the containment-container relationship is expressed through the sign. This function will be later used to select which is the highest child, from which start the GSM visiting task, thus allowing to choose which element allows the visit most of the contained objects by reducing the visiting steps[1].

▸ **Definition 31** (Heights). *Given a nesting loop fee GSM $g = (o, O, \ell, \xi, \phi)$, the **relative height**[2] of two objects $o', o'' \in O$ is expressed as the function returning n if o' contains o'' after maximum[3] n applications of φ over o' (5.1), $-n$ if o' is contained by o'' after n applications of φ over o'' (5.2), or zero if the objects are the same or if they are siblings (5.3). For all the other cases (e.g., if they have a common ancestor), the function is undefined.*

$$rh(o', o'') = \begin{cases} \max S + 1 & S = \{\, n > 0 \mid o'' \in \varphi^n(o') \,\}, S \neq \varnothing \quad (5.1) \\ -(\max S) - 1 & S = \{\, n > 0 \mid o' \in \varphi^n(o'') \,\}, S \neq \varnothing \quad (5.2) \\ 0 & o' = o'' \vee \exists o''' \in O.\, o', o'' \in \varphi(o''') \wedge o' \neq o''' \wedge o'' \neq o' \quad (5.3) \end{cases}$$

The former definition can be also used to sort[4] the children $\varphi(o)$ of one single containing object o via their relative height by using the following function[5]:

$$h_o(o') = \max \left\{\, rh(o', o'') \neq 0 \;\middle|\; o'' \in \varphi^*(o) \wedge o'' \neq o' \,\right\}$$

Therefore, we can generalize such function to the height of the whole GSM as follows[6]:

$$h(g) = \max \left\{\, rh(o, o') \geq 0 \;\middle|\; o' \in \varphi^*(o) \,\right\}$$

◂

We can also define the concept of "strongly nested object-set" by restating the "strongly connected component" for graphs over the φ containment function. In this case, we want to select from O of a GSM g the sets that contain both the containers and their contents, recursively towards their respective leaves.

▸ **Definition 32** (Strongly nested object-set). *Given a GSM $g = (o, O, \ell, \xi, \phi)$, it is said that $O' \subseteq O$ is a **strongly nested object-set** of g if each object in O' is either a container or a content of another different object of O' ($\forall u, v \in O'. u \neq v' \Rightarrow u \in \varphi^+(v) \vee v \in \varphi^+(u)$).*

*We also say that O' is a **strongly nested object-set component** of g if $O' \subseteq O$ and if O' is a maximal strongly nested object-set: this means that it does not exist any object-set O'' such that $O' \subset O'' \subseteq O$ and O'' is a strongly nested object-set.*

*If we also fix a subset $\delta O \subseteq O$, then we can also say that O' is a **strongly nested object-set (component) with respect to** δO if O' is a strongly nested object-set component of O such that $O' \subseteq \delta O$. Therefore, δO provides an upper bound for O'.* ◂

Given that the concept of "subset" is present both in relational data (via set theory) and on property graphs (via the "subgraph" definition from graph theory), we may also provide a similar concept for our GSM model. We're going to use later on this concept for "filtering" operators, allowing a partial extraction of the given data structures. We can also use the notion of "subgraph" or "subset-of" for GSMs and define it as follows:

[1] For a more practical definition, see line 449 of the code presented in Appendix A on page 225 in OCaml.
[2] See line 407.
[3] Please note that, given that the GSM is nesting loop free, it there exists an $n > 0$ such that $\varphi^m(o) = \varnothing$ for any $o \in O$ and $m \geq n$.
[4] See line 449.
[5] See line 443.
[6] See line 434.

▸ **Definition 33** (Substructure). *Given two GSMs $g = (o, O, \ell, \xi, \phi)$ and $g' = (o', O', \ell', \xi', \phi')$, it is said that g' is a* ***substructure*** *of g ($g' \subseteq g$) if and only if it exists a transcoding function $\varsigma(o_{c+1}) = o_c$, usually called* ***morphism*** *in graph literature, mapping each object in O' to one single object in O, where each pair of correspondent objects o_{c+1} and o_c maintain the same labels and expressions, but their containment functions are one the subset of the other:*

$$\begin{aligned}\forall o \in O'.\ell'(o) = \ell(\varsigma(o))\wedge \\ \xi'(o) = \xi(\varsigma(o))\wedge \\ \forall p \in \mathcal{L}.\varsigma(\phi'(o,p)) \subseteq \phi(\varsigma(o),p)\end{aligned}$$

When the transcoding function is made explicit, we denote the substructure relation as $g' \subseteq_\varsigma g$. ◂

From this basic definition, we can infer the definition of GSM equivalence, where objects' labels, expressions and containments are compared independently from the id appearing in O.

▸ **Definition 34** (GSM Equivalence). *Given two GSMs $g = (o, O, \ell, \xi, \phi)$ and $g' = (o', O', \ell', \xi', \phi')$, it is said that g' abd g are equivalent ($g' \equiv g$) if and only if it exists a bijective transcoding function ς through which $g' \subseteq_\varsigma g$ and $g \subseteq_{\varsigma^{-1}} g'$.* ◂

5.1.1 script, a MetaModel for GSM

We now describe the MetaModel for the GSM model; script[7] is an untyped imperative language with dynamic scope and shallow binding, where each program is represented by a list of expressions which are lazy evaluated. Each program returns the value resulting from the evaluation of the last statement. The script syntax is depicted in Listing 5.1 on the following page: script provides some native operations over numbers (either bignum integers or doubles, $\mathbb{Z} \cup \mathbb{Q}$), strings, booleans and lists. All those types are considered compatible between each others, that is they always allow a conversion into a specific representation. Therefore, this language aims to define transcoding functions that can be used for object transformations.

This language adopts the implicit casting while evaluating the native operations. As a consequence, the number add operator + is implemented as a binary function + taking a pair of numbers as an input and returning another number:

$$+ : \mathbb{Z} \cup \mathbb{Q} \times \mathbb{Z} \cup \mathbb{Q} \to \mathbb{Z} \cup \mathbb{Q}$$

Consequently, the following expression:

$$\texttt{expr}_1 \ \texttt{+} \ \texttt{expr}_2$$

will be interpreted[8] as follows:

$$[\![\texttt{expr}_1 + \texttt{expr}_2]\!]_\varnothing = \textit{toNumeric}([\![\texttt{expr}_1]\!]_\varnothing) \ + \ \textit{toNumeric}([\![\texttt{expr}_2]\!]_\varnothing)$$

where $\varnothing$ represents the environment Γ with which such expressions are evaluated; this means that for the moment, Γ does not influence the outcome of the expression's evaluation.

[7] See http://rebrand.ly/unibo41251 for the source code implementing such language. Within this project, you can see the actual implementation of some functions that will follow, such as *toNumeric*, *toFunction toList* that are going to be used within the following denotational semantics.

[8] A complete discussion of script's denotational semantics [NN92] goes beyond the goals of the present thesis. In brief, we use the notation $[\![P]\!]_\Gamma$ instead of the less intuitive $\Gamma[\![P]\!]$.

Listing 5.1 Subset of the script language in Antlr4.

```
script  : (expr ’;’)* expr ;

expr : ’(’ expr ’)’                    #paren
| expr ’+’ expr                        #number add
| expr ’-’ expr                        #number subtract
| expr ’/’ expr                        #number divide
| expr ’*’ expr                        #number multiply
| expr ’++’ expr                       #string concatenation
| expr ’@’ expr                        #list append
| expr ’&&’ expr                       #boolean and
| expr ’||’ expr                       #boolean or
| ’not’ expr                           #boolean negation
| expr ’==’    expr                    #equals
| expr ’!=’    expr                    #not equals
| expr ’<=’    expr                    #less equal
| expr ’>=’    expr                    #greater equal
| expr ’>’     expr                    #greater
| expr ’<’     expr                    #less
| expr ’:=’    expr                    #assignment
| expr ’.’     expr                    #method invocation
| ’(’ expr  expr ’)’                   #apply right element to left function
| expr ’=>’ expr                       #if left is true when bool evaluated,
                                       # return it, otherwise return right
| ’if’ expr ’then’ expr ’else’ expr    #if ... then ... else
| ’substring(’ expr ’,’ expr ’,’ expr ’)’ #substring (or sublist of first,
                                       #between the second and third index)
| expr ’[’ expr ’]’                    #get the element indexed with right
    ↪ in left
| expr ’[’ expr ’]:=’ expr             #put the rightmost expression
| expr ’in’ expr                       #contains
| ’remove’ expr ’from’ expr            #remove left element from right list
| EscapedString                        #native string
| BOOL                                 #native boolean
| NUMBER                               #native number
| ’{’ (expr ’,’)* expr ’}’             #native expressions’ array
| VARIABLE ’->’ ’{’ (expr ’;’)* expr ’}’#lambda function
| VARIABLE                             #variable
| ’map(’ expr ’:’ expr ’)’             #map left list over right function
| ’select(’ expr ’:’ expr ’)’          #select left elements using right
    ↪ pred.
;

BOOL : ’tt’ | ’ff’ ;
VARIABLE    : [a-z]+ ;
EscapedString : ’"’ ( ’""’ | ~["\r\n] )* ’"’;
NUMBER : [0-9]+ (’,’[0-9]+)? ;
```

Such implicit casting operations allow to reduce the number of the data-type specific operations, such as defining the length of the list,that can be expressed as outlined in the following example.

▸ **Example 23.** *We want to evaluate the following* `script` *expression:*

$$\texttt{0 + \{1, 2, 3\}}$$

where a bignum `0` *is added to a list*[9] `{1, 2, 3}`*. Given that our canonical function toNumeric maps each list to its length expressed as bignum integers, the following expression will be evaluated to* 3.

$$\begin{aligned}[\![\texttt{0 + \{1, 2, 3\}}]\!]_\varnothing &= \mathit{toNumeric}([\![\texttt{0}]\!]_\varnothing)\ +\ \mathit{toNumeric}([\![\texttt{\{1, 2, 3\}}]\!]_\varnothing)\\ &= \mathit{toNumeric}(0)\ +\ \mathit{toNumeric}([1,2,3])\\ &= 0+3=3\end{aligned}$$

A similar strategy is used for the definition of the `map` and filter operator (`select`). The first operator has the following associated semantics:

$$[\![\texttt{map(expr}_1\texttt{ : expr}_2\texttt{)}]\!]_\Gamma = [\mathit{toFunction}([\![\texttt{expr}_2]\!]_\Gamma)(x) \mid x \in \mathit{toList}([\![\texttt{expr}_1]\!]_\Gamma)]$$

In particular, *toFunction* returns a function if the input is a function itself, or acts as a constant function otherwise. When the first argument is a list, `map` acts as a mapping function over the list, otherwise if it is a function acts as a concatenation between functions, otherwise if it is a single value, it acts as a function application. Similar considerations can be carried out for `select`.

A `script` program allows to define variables and functions, which declarations creates an association within an environemnt Γ. Such environment is a set of variable-expressions associations, that can be updated or defined for the first time with an assignment operation:

$$\texttt{VARIABLE := expr}$$

Therefore, we have that evaluation of such expression will result into an update of Γ as follows:

$$[\![\texttt{VARIABLE := expr; c}]\!]_\Gamma = [\![\texttt{c}]\!]_{\Gamma\cup\{(\texttt{VARIABLE},\texttt{expr})\}}$$

The `script` language is also capable of defining recursive functions, through which it is possible to generate numeric sequences, and hence generate enumerations. In particular, the following expression will generate a set containing all the positive natural numbers from 2 to 5:

```
f := x -> { if (x >= 5) then x else (x @ (f (x+1)))};
(f 2)
```

This language even allows to emulate the result of a fold operation over a set. The following example shows how we can add all the natural numbers from 1 to 10 by using the following expression:

[9] Please observe that the `script` notation for a list is `{. . . }`, while the one used for the lists within the GSM model is [. . .]. In this way we can distinguish between the syntactic representation and its semantic one.

```
acc := 0;
sgen := x -> { if (x >= 10) then x else (x @ (sgen (x+1))) };
s := (sgen 1);
select( s : x -> { acc := (x + acc) ; ff } );
acc
```

In particular, this program initializes the accumulator acc to zero, defines a set containing all the numbers from 1 to 10 and stores it into s, then uses select to iterate over s and produce an empty result while, at each iteration step, acc is incremented by one of the elements contained within s. Last, the result of the accumulation is returned.

Consequently, we can generally define the foldl operator as follows:

```
foldl := x -> {
        acc := x[0];
        select(x[1] : y -> {
                acc := (x[2] {y, acc});
               ff
        });
        acc
}
```

Before defining any possible operation over the lists, we must make explicit the meaning of set operations over lists.

▸ **Definition 35** (Set operations over lists). *Given a binary set operator* $\bowtie$*, its definition over lists is defined as follows:*

$$L_1 \bowtie L_2 = \texttt{distinct}(L_1 \bowtie L_2, =)$$

where distinct *returns the element where the repeated elements are preserved, and only the leftmost instance is kept*[10]*:*

```
let rec distinct l eq =
        let rec removehead x ls =
        match ls with
        | [] -> []
        | h::t -> if (eq h x) then t else h::(removehead x t)
in match l with
| [] -> []
| a::[] -> [a]
| a::b -> a::(removehead a (distinct b))
```

◂

5.1.1.1 Using environment Γ with external GSM values

An interesting feature of such language is its possibility of initializing Γ with external objects prior to the evaluation of the expression: this feature is crucial to manipulate and

[10] This operator can be also defined in script as:
`distinct = x ->{fold [x, y ->{if (y[0] in y[1])then y[1] else [y[0]] ++y[1]}]}`

define predicates over the labels, expressions and containments associated to single objects. For this reason `script` will reserve variable names "o" and "g" respectively to the current GSM object of interest represented as an object-oriented `Object`, and to the reference object of the current GSM. Given a GSM (o, O, ℓ, ξ, ϕ), we can represent each object $o' \in O$ as an object in a object oriented language. In particular, this[11] is the object representation that is used in `script` for the representation of object as external objects.

```
class Object {
        id objectId;
        public Object(id i) { objectId = i; }

        private List phiFuncs(boolean toObject) {
                List toreturn = new List();
                for (ℒ e : dom(ϕ(objectId))) {
                        List elist = new List();
                        elist.add(e);
                        List idlist = new List();
                        for (id i : ϕ(id,e)) {
                                idlist.add(toObject ? new Object(
                                   ↪ i) : i);
                        }
                        elist.add(idlist);
                }
                return toreturn;
        }

        public id   id()       { return objectId; }
        public List ell()      { return ℓ(objectid); }
        public List xi()       { return ξ(objectId); }
        public List phi()      { return phiFuncs(true); }
        public List phiVisit() { return phiFuncs(false); }
}
```

This object will use two different phi implementations: `phi` can be used to manipulate the containment function, and hence will return the GSM objects as `id`s, while `phiVisit` can be used in depth visiting operators for the GSM, and hence will return the GSM objects as `Object`s. Last, a GSM is represented by its reference object as an `Object` `g`.

▸ **Example 24.** *Suppose to have an object $o = 2345$ within a GSM (o, O, ℓ, ξ, ϕ), where o is associated to the following values:*

$$\ell(o) = [\text{"ciao1"}, \text{"ciao2"}, \text{"ciao3"}] \qquad \xi(o) = []$$

$$\phi(o, \text{"elemento"}) = [0,1,2] \qquad \phi(o, \text{"elementu"}) = [3,4,5]$$

In particular, we initialize Γ by associating "`o`*" and "*`g`*" to the same object o. In particular, we can get its id by invoking the* **`id`** *method as follows:*

$$[\![\texttt{o}.\textbf{id}]\!]_\Gamma = [\![\texttt{o}]\!]_\Gamma.\textbf{id}() = 2345$$

[11] `http://rebrand.ly/unibobd4f8` provides a more complete representation of the code that follows.

Similarly, finite functions as $\phi(o)$ can be expressed by their graph. Even in this case we can use the method invocation to return the content associated to $\phi(o)$ as a list of lists containing two elements, where the first is an expression e and the second is the content of $\phi(o,e)$.

$$[\![\texttt{o.}\textbf{phi}]\!]_\Gamma = [[\text{“elemento”}, [0,1,2]], [\text{“elementu”}, [3,4,5]]]$$

Consequently, from now on we choose to graphically represent the graph of a function f as a list of pairs (represented as lists of two elements), where each first element represents an element x from the domain, and the second element represents the values associated to the codomain: if $f(x)$ is a function for some $x \in \mathrm{dom}(f)$, $f(x)$ can be represented in `script` *with the same list representation. This also means that we can easily express the function* dom *for functions represneted by their graph as follows:*

```
dom := f -> { map(f : x -> {x[0]})}
```

Then, we can use a combination of `select` *and* `get` *list accessors to obtain the values $\phi(o, \text{“elemento”})$ by selecting the expected list position:*

$$\begin{aligned}
&[\![\texttt{(select((o.}\textbf{phi}\texttt{) : x -> \{ x[0] ==}\ \text{“elemento”}\texttt{\}))[0] [1]}]\!]_\Gamma\\
&= [\![\texttt{(select((o.}\textbf{phi}\texttt{) : x -> \{ x[0] ==}\ \text{“elemento”}\texttt{\}))}]\!]_\Gamma\texttt{.}\textbf{get}\texttt{(0).}\textbf{get}\texttt{(1)}\\
&= [x \in [\![\texttt{o.}\textbf{phi}]\!]_\Gamma \mid \texttt{x.}\textbf{get}\texttt{(0) ==}\ \text{“elemento”}\ ]\texttt{.}\textbf{get}\texttt{(0).}\textbf{get}\texttt{(1)}\\
&= [x \in [[\text{“elemento”}, [0,1,2]], [\text{“elementu”}, [3,4,5]]] \mid \texttt{x.}\textbf{get}\texttt{(0) ==}\ \text{“elemento”}\ ]\texttt{.}\textbf{get}\texttt{(0).}\textbf{get}\texttt{(1)}\\
&= [[\text{“elemento”}, [0,1,2]]]\texttt{.}\textbf{get}\texttt{(0).}\textbf{get}\texttt{(1)}\\
&= [\text{“elemento”}, [0,1,2]]\texttt{.}\textbf{get}\texttt{(1)}\\
&= [0,1,2]
\end{aligned}$$

In particular, we can use the following shorthand for accessing such values:

```
e[s] ≝ (select( e :x ->{ x[0] ==  s}))[0] [1]
```

so that we can return the “elemento” Consequently, we can write an expression allowing us to map, filter and extend the containment. In particular, the following expression will multiply each id referenced by “elemento” for ϕ by 2, remove the “elementu” elements and create a new containment “neu” containing 1. The desired expression is the following one:

```
{{"elemento", map( (o.phi) ["elemento"] : x -> { (x * 2) } )}} @
    ↪ {{"neu", {1}}}
```

Last, by using the foldl function previously defined, we can also define φ^+ within script as follows:

```
varphiplus := x -> {
        (foldl {{}, x.phi, y -> {y[0][1] @ y[1]} })
        @
        (map(x.phi : varphiplus))
}
```

Appendix A on page 225 also provides a functional definition of such GSM model through the usage of a functional programming language, OCaml. Moreover, the use case in Section 5.4.1 on page 139. and 6.3.3 on page 182. will provide some examples of how such Γ environment for GSM values achieves structural aggregations.

5.1.2 Characterizing object identifiers

Before introducing either the operations that are possible on top of this model or the nested graph data model itself, we must consider if we have to provide some further constraints on object identifiers. In particular, we must assume that each identifier maps to one single and possible data representation. The same concept was introduced with the Object Identity problem at Section 3.1.2 on page 60, where the Skolem functor $f_{\mathfrak{R}}$ always provided an unique mapping between object and its id. Similarly to other presented graph model, we adopted the symmetrical solution, that is the index is mapped to the data and not vice versa. GSM already provides this constraint because ϕ-containment is defined by the list of object identifiers associated with each object's property.

An object transformation should always generate a different object with a different id, and shall not update the original object (uniquely referenced by its id) with newly associated values. The reason is twofold: we want that shared nothing distributed algorithms make local changes consistent with the global status of the computation, and that the implementations of GSM over object oriented languages maintain the same object reference over the same already-existing object. This constraint must be also satisfied by the data operations that manipulate the objects.

In order to do so, we must ask ourselves what is the aim of querying data, and which is a suitable representation of the answer to such query within the data model. If we take SQL as a reference, our final query language must allow the creation of views on top of the database (in our case, the *GSM* itself), without necessarily modifying the underneath data representation to which the result is subjected. The same concept is self evident within the relational algebra: the relations took as an input by the algebraic expression are manipulated within the sequence of operations, and returned as a new relation. Consequently, at each step the data subsumes some changes, either structurally or from the informative content, and new elements are created after each operation.

Given that the creation of new data within queries implies the creation of new identifiers, it is important that even the data model guarantees the creation of identifiers that are easily referable to a previous computation [Ber14] for optimizing some data operations. This feature is going to be crucial for the algorithm that will be used for graph nesting in Chapter 7 on page 195. In particular, each element j represented at the i-th step of computation, should change its id jointly with its internal representation. In order to do so we're going to use dovetailing functions [Odi92] that associate unique integer numbers to a given pair of numbers:

▸ **Definition 36** (Dovetailing function). *Given a pair of two integers $i, j \in \mathbb{N}$, the* ***dovetailing function*** *associates to them an unique integer $j_i \overset{\text{def}}{=} dt(i,j) \in \mathbb{N}$ defined by the following function:*

$$j_i \overset{\text{def}}{=} dt(i,j) = \frac{(i+j)(i+j+1)}{2} + j$$

This can be also showed by the definition of the following ***inverse dovetail function*** *$dt^{-1}(j_i)$ [Ber14]:*

$$dt^{-1}(j_i) = (dt^{-1}_{\texttt{Left}}(j_i), dt^{-1}_{\texttt{Right}}(j_i))$$

where each single component is defined as follows:

$$dt^{-1}_{\texttt{Right}}(j_i) = j_i - {}^1\!/_2 \cdot \left(\left\lfloor {}^1\!/_2 \cdot \left(\sqrt{8j_i+1} - 1 \right) \right\rfloor + 1 \right) \cdot \left\lfloor {}^1\!/_2 \cdot \left(\sqrt{8j_i+1} - 1 \right) \right\rfloor$$

$$dt^{-1}_{\texttt{Left}}(j_i) \overset{\text{def}}{=} \left\lfloor {}^1\!/_2 \cdot \left(\sqrt{8j_i+1} - 1 \right) \right\rfloor - dt^{-1}_{\texttt{Right}}(j_i)$$

◂

In particular, to each element j represented at the i-th computational step, the computation shall have an id $dt(i,j)$, and $i = 0$ when the object is kept unaltered from the data representation level. When one of the two numbers has a maximum value (e.g., when the number L of the query operators involved within the computation is known), then we can reduce the size of the generated number as follows:

▸ **Definition 37** (L-bounded dovetailing function)**.** *Given a pair of two integers $i,j \in \mathbb{N}$ where i is upper bounded by L, the L-**bounded dovetailing function** associates to them an unique integer $dt(i,j) \in \mathbb{N}$ defined by the following function:*

$$dt^L(i,j) = j \cdot (L+1) + i$$

◂

Such dovetailing function can be also used to uniquely associate a number to a list of numbers. In such contexts we can use the dtl function:

$$dtl(l) = \begin{cases} 0 & l = [] \\ 1 + dtr(|l|, dtr(l)) & \text{oth.} \end{cases} \tag{5.4}$$

where dtr is recursively defined as follows:

$$dtr(l) = \begin{cases} e & l = e :: [] \\ dt(h, dtr(t)) & l = h :: t \end{cases}$$

This other function will be later on used when new elements will be generated from scratch without necessarily transforming the already-existing ones.

Dovetailing over object identifiers: properties

Suppose that each object with id y within the original data sources is represented by an unique id $dt(0,y) = y_0$ and that, after each transformation step, y_0 is transformed into a new object y_1. Hereby, given the object y at the n-th computational step with id y_n, we would like to know if there is an efficient way to compute y_{n+1} from y_n: the attribution of such step numbers can be given by the query system by the query plan engine by enumerating all the query computation steps. After observing that the following lemma is true (proof at page 233):

▸ **Lemma 5.1.1.** $dt(x+1,y) = dt(x,y) + x + y + 1$

we can observe that:

$$dt(x+2,y) = dt(x,y) + 2(x+y) + 1 + 2$$

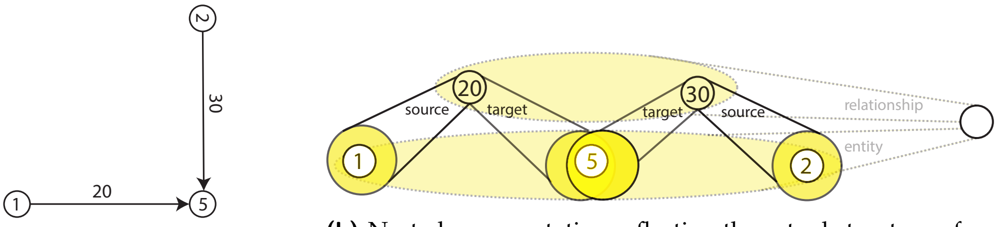
and hence, we can prove the following lemma by induction (proof at page 233)

▸ **Lemma 5.1.2.** $dt(x+i,y) = dt(x,y) + i(x+y) + \sum_{n=0}^{i} n$

Hereby, we can express an arbitrary increment i of the computational step transforming y by using an additive step, where $y_{n+1} = y_n + y + 1$ and that $y_{n+1} = y_0 + (n+1) \cdot y + {}^{(n+1)(n+2)}/_2$. This implies that the transformation of any object y_n into the next computational step ς^{n+1} can be expressed as a view ς over the object y_n producing a new object $\varsigma(y_n) = y_{n+1}$. As a result, we can avoid to replicate the information generated at each computational step.

We can also prove a lemma similar to 5.1.1 that is going to be used for object creation:

▸ **Lemma 5.1.3.** $dt(x,y+1) = dt(x,y) + x + y + 2$



(a) Traditional representation of the graph $(\{1,2,5\},\{\,30=(2,5),20=(1,5)\,\})$.

(b) Nested representation reflecting the actual structure of the data model.

Figure 5.2 Nested graph representation (b) of a traditional graph data structure extended with edge id (a).

5.2 Nested Graph

Despite the definition of efficient graph distributed models using a nested relational representation [LBO+15], a proper graph data model embodying a nested representation of graphs is missing. As we already saw, GSM allows to perform arbitrary nestings up to user-defined model constraints. Now, we must specialise this data model to distinguish the objects representing vertices from the ones representing edges; moreover, we want also that objects representing atoms should be distinguishable from the other two. The reason for doing this is that different kind of operations can be performed simultaneously over vertices and edges, as presented by the join operator in the last chapter. Therefore, instead of representing vertices and edges as belonging to the same set and then extract them in order to perform a distinct set of operations, we will directly distinguish the vertices and edges as belonging to two distinct properties of the same (reference) object.

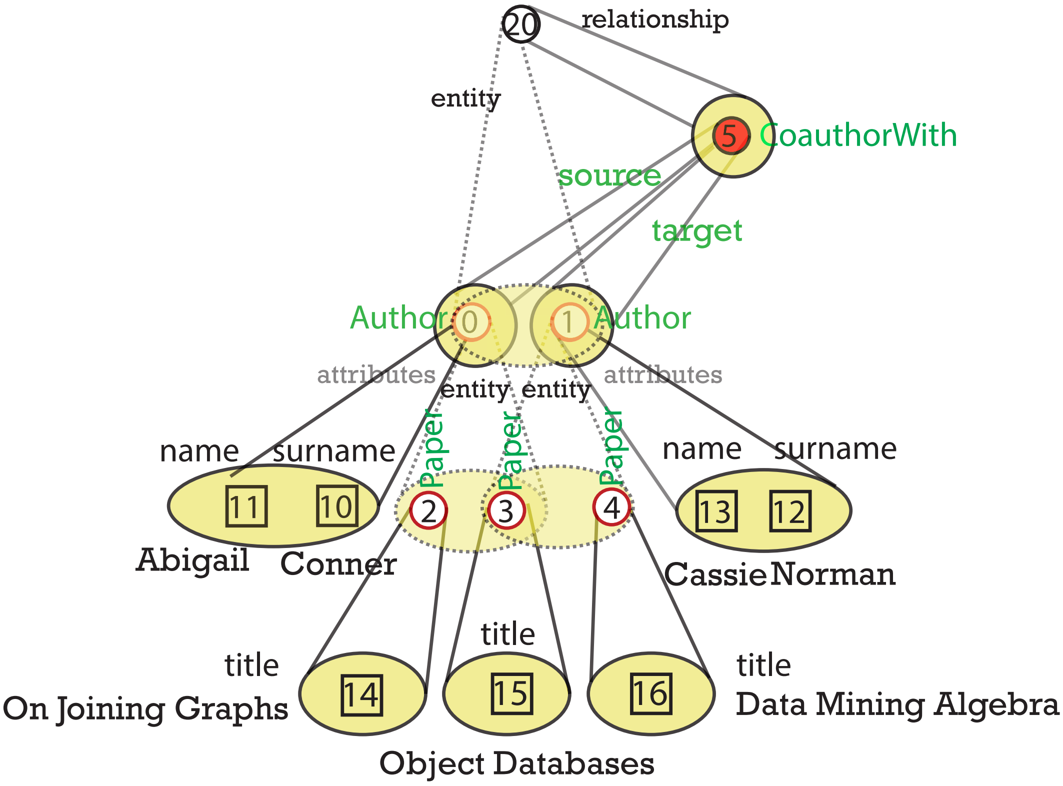
▸ **Definition 38** (Nested Graph). *A **nested graph** is a nesting loop free GSM $\eta=(\mathfrak{g},O,\ell,\xi,\phi)$ where $\mathfrak{g}$ is an object containing both the vertex ($\phi(\mathfrak{g},\text{“Entity”})$) and edge ($\phi(\mathfrak{g},\text{“Relationship”})$) set. Each object $g\in\varphi^*(\mathfrak{g})$ such that “Relationship” $\in \mathrm{dom}(\phi(g))\wedge$ “Entity” $\in\mathrm{dom}(\phi(g))$ is said to be a **graph**. Each graph must satisfy the following properties:*

- *Each vertex v' in $\mathfrak{g}$ is associated to a “Entity”collection ($\exists g\in\varphi^*(\mathfrak{g}).v'\in\phi(\mathfrak{g},\text{“Entity”})$).*
- *Each edge e' in $\mathfrak{g}$ is associated to a “Relationship”collection ($\exists g\in\varphi^*(\mathfrak{g}).e'\in\phi(g,\text{“Relationship”})$).*
- *The same object shall not belong to both “Entity”or “Relationship”collections of any object*[12] *($\bigcup_{o\in\varphi^*(\mathfrak{g})}\phi(o,\text{“Entity”})\cap\bigcup_{o\in\varphi^*(\mathfrak{g})}\phi(o,\text{“Relationship”})=\varnothing$).*
- *All the edge elements e' must contain only two vertices s and t such that s is a source for e' ($\phi(e',\text{“src”})=[s]$) and t is its target*[13] *($\phi(e',\text{“dst”})=[t]$). Moreover, s and t must both be vertices ($\exists g,g'\in\varphi^*(\mathfrak{g}).s\in\phi(g,\text{“Entity”})\wedge t\in\phi(g',\text{“Entity”})$). Such elements are uniquely identified by a λ function which is defined as follows:*

$$\lambda(o)\overset{\text{def}}{=\!=}\phi(o,\text{“src”})\times\phi(o,\text{“dst”})$$

[12] Please note that, by enforcing the model with such constraint, no RDF graph representation will be possible. On the other hand, alignment over the edges' types would not be formulated. As we will see on the later sections, we will not strictly perform such alignments at the nested graph level, but we will use general *GSM* to do so. Nevertheless, the thesis will focus only on this nested graph model.

[13] More formally, $\forall o\in O.\forall e'\in\phi(o,\text{“Relationship”}).|\phi(e',\text{“src”})|=1\wedge|\phi(e',\text{“dst”})|=1$. Please note that, by relaxing such constraint to an arbitrary non-zero number of elements (that is, $s\in\phi(e',\text{“src”})$ with $|\phi(e',\text{“src”}|\geq 1$ and $t\in\phi(e',\text{“dst”})$ with $|\phi(e',\text{“dst”})|\geq 1$), this representation allows to represent hypergraphs, that are graph representations where edges allow more than one possible source and edge.



■ **Figure 5.3** Representation of a nested graph in different steps. This data structure represents "Author Abigail Conner is Coauthor with Cassie Norman because they both authored the Paper "Object Databases". Abigail Conner also wrote "On Joining Graphs", while Cassie Norman wrote "Data Mining Algebra". From now on we will represent the objects with empty containments as squares, and the non-empty ones as circles. Edges are represented by circles filled in red, while vertices are represented by red bordered circles. Moreover, labels are placed above the nodes, while the expressions are placed below.

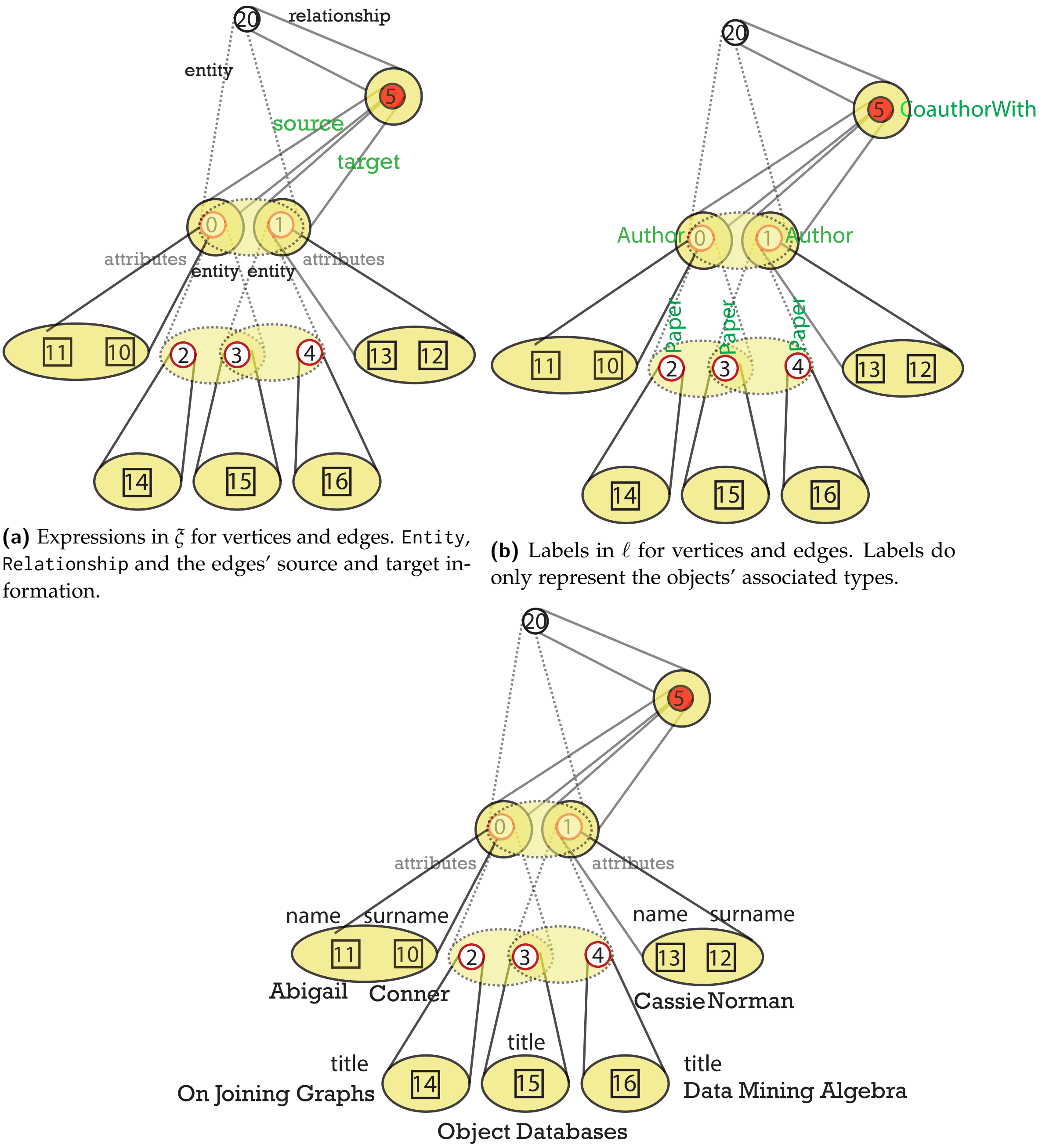
Given that are not specific on what a 'graph' should be, both vertices and edges may represent (nested) graphs. ◂

The proposed nested graph data model provides a "(nested) relational representation" of a graph: vertices and edges are both represented as objects, and edges' source and target vertices are represented as edges' fields. Our model allows to nest (sub)graphs for each vertex and edge: this is due to the fact that an object can contain "Entity" and "Relationship" properties. Consequently, this data structure allows to represent traditional graphs and multigraphs, as outlined in Figure 5.2 on the previous page. The actual transformation function allowing to transform a graph into a nested graph representation will be provided in Section 5.3 on page 136, alongside with other data transformation representations. The example that follows provide some insights on how such data structures can be used to represent nested components.

▸ **Example 25.** *Figure 6.3 provides a graphical representation of a nested graph.*

$$N = (20, \{0, 1, \ldots, 4, 10, \ldots, 16, 20\}, \ell, \lambda, \xi, \phi)$$

As we can see, the vertex set is $\phi(20,$"Entity"$)$ *while the edge set is* $\phi(30,$"Relationship"$)$.



(a) Expressions in ξ for vertices and edges. `Entity`, `Relationship` and the edges' source and target information.

(b) Labels in ℓ for vertices and edges. Labels do only represent the objects' associated types.

(c) Labels and expressions for the attributes, that are objects containing neither ``` ``Entity'' ``` nor ``` ``Relationship'' ``` expressions. Their keys are expressed in ℓ while their values are stored as ξ expressions.

Figure 5.4 Analysing each component in Figure 5.3 on the facing page.

Consequently, those sets are represented as objects providing the following collections (i.e., lists):

$$\phi(20,\text{“Entity”}) = [0,1] \qquad \phi(20,\text{“Relationship”}) = [5]$$

Given that both vertices and edges are represented as objects, vertices and edges can contain other vertices and edges, too. Moreover, those objects can contain other objects, too, such as attributes, or the information of the edges' source and target vertices. In particular, such nested graph contains two authors, 0 and 1, which are coauthors. In particular, the coauthorship relation is represented by the edge 5, which contains source and target elements as nested nodes:

$$\ell(0) = [\texttt{Author}] = \ell(1) \qquad \ell(5) = [\texttt{CoAuthorship}]$$

$$\phi(5,\text{“src”}) = [0] \qquad \phi(5,\text{“dst”}) = [1]$$

Moreover, each vertex nests the information of the papers that have been published by the containing author. Among this information, their names and surnames are also contained:

$$\ell(2) = \ell(3) = \ell(4) = [\texttt{Paper}]$$

$$\phi(0,\text{“Entity”}) = [2,3] \quad \phi(0,\text{“Attribute”}) = [11,10]$$

$$\phi(1,\text{“Entity”}) = [3,4] \quad \phi(0,\text{“Attribute”}) = [13,12]$$

$$\ell(11) = \ell(13) = [\texttt{Name}] \qquad \xi(11) = [\text{“Abigail”}] \quad \xi(13) = [\text{“Cassie”}]$$

$$\ell(10) = \ell(12) = [\texttt{Surname}] \quad \xi(10) = [\text{“Conner”}] \quad \xi(12) = [\text{“Norman”}]$$

For each paper, the title information is also provided through object containment:

$$\ell(14) = \ell(15) = \ell(16) = [\texttt{Title}]$$

$$\xi(14) = [\text{“On Joining Graphs”}] \quad \xi(15) = [\text{“Object Databases”}] \quad \xi(16) = [\text{“Data Mining Algebra”}]$$

Please note that this graphical representation is verbose, and does clarify which object is a vertex and which element is an edge only by its containment into a graph object. We call **simple vertex** (**simple edge**) a vertex (edge) which is not a graph. A more interesting example of Nested Graphs is going to be provided by Example 26 on page 141.

5.3 Data model translation functions

We now define the τ operators for translating some of the previous models into either GSMs or Nested Graphs in a purely syntactic fashion, thus providing a common data representation required by the Global as A View scenario as outlined in Equation 2.5 on page 50. Please also note that τ cannot be represented in a fixed query language, because it may translate any possible present (and future) data structure. For this reason, we are going to implement τ using some generic pseudocode. This section will focus on some τ definitions showing that the nested graph data structure allows to represent all the aforementioned structured and semistructured representations and the GSM model.

In particular, we're going to use the usual notation for function overriding [NNH05] that has already been presented in Definition 23 on page 96 with the $\oplus$ notation. In particular, $f(x) := y$ is a shorthand for $f \oplus [[x,y]]$, where $[[x,y]]$ is the graph[14] *of the function $\{x\} \to \{y\}$ mapping x into y.*

[14] The graph of a function is the collection of all the ordered pairs represented as a list $[x, f(x)]$ for each $x \in \mathrm{dom}(f)$. We will use this notation for reasons that will be clear in Section 5.1.1.1 on page 128.

Algorithm II.4 Relational Table (τ_R) and Database (τ_{DB}) to GSM

```
1:  $\ell \overset{\text{def}}{=}$ **new func.** $\varnothing \to \wp(M)$;
2:  $\xi \overset{\text{def}}{=}$ **new func.** $\varnothing \to \wp(\mathcal{L})$;
3:  $\phi \overset{\text{def}}{=}$ **new func.** $\varnothing \to \mathbb{N}$;
4:
5:  **function** $\tau_R(\, r(R), \ell, \xi, \phi, seed\,) : GSM$        ▷ $R = (A_1 \dots A_n)$, $seed \geq 1$
6:      $O \overset{\text{def}}{=} \{\, dt(seed, dt(i,j))_0 \mid 1 \leq i \leq |r|,\ 1 \leq j \leq n \,\} \cup \{dt(seed, dt(0,0))_0\}$
7:      $r_o \overset{\text{def}}{=} dt(seed, dt(0,0))_0$      $\phi(r_o, \text{“Entity”}) := [dt(seed, dt(i,0))_0 | 1 \leq i \leq |r|]$      $\ell(r_o) := [\text{‘r’}]$
8:
9:      **for** $t_i \in r$ **s.t.** $i \leq |r|$ **do**
10:         $o \overset{\text{def}}{=} dt(seed, dt(i,0))_0$
11:         $\ell(o) := [\text{‘R’}]$;      $\phi(o, \text{“Attribute”}) := [dt(seed, dt(i,j))_0 | 1 \leq j \leq n]$
12:         **for** $\forall A_j \in R$ **do**
13:             $\ell(dt(seed, dt(i,j))_0) := [\text{‘‘}A_j\text{’’}]$      $\xi(dt(seed, dt(i,j))_0) := [t[A_j]]$
14:     **return** $(r_o, 0, O, \ell, \xi, \phi)$
15:
16: **function** $\tau_{DB}(\, DB, seed\,) : GSM$
17:     $db \overset{\text{def}}{=} seed$;      $\ell(db) := [DB]$;      $\phi(db, \text{“Entity”}) := [dt(i + seed, dt(i,j))_0 | 1 \leq i \leq |DB|]$
18:     $O \overset{\text{def}}{=} \{db\}$
19:     **for each table** $r_i(R_i) \in DB \equiv \{\, r_1(R_1), \dots, r_n(R_n) \,\}$ **do**
20:         $(r_i, O_i, \ell, \xi, \phi) := \tau_R(r_1(R_1), \ell, \xi, \phi, i + seed)$
21:         $O := O \cup O_i$
22:     **return** $(db, 0, O, \ell, \xi, \phi)$
```

Relational Databases to GSM

We now propose two translation functions, one for transforming relational tables to GSMs, and the other one for translating a full relational database into GSM. Such GSM are (nested) graphs composed of entities with no relationships. Therefore, it implies that such model can be used to represent a whole multidimensional database. For these first translation functions we choose to translate instances of the relational model (either tables or whole relational databases) into nested vertices (entities) belonging to one single nested object: in Section 3.1.2 on page 59 we discussed that relations can be used to define either entities or relationships with a semantic overload. Given that such distinction is model dependant, we leave to the single user the definition of a function that translates such GSM into a faithful representation of the ER model. Algorithm II.4 provides at line 5 the associated τ_R function returning a GSM for each relation table $r(R)$. In particular, the *seed* argument is required for generating new elements with distinct ids. For each table $r(R)$ with schema R, we want to return an object for each tuple in t, nesting other vertices which provide the information stored inside each field. Each object obtained from the relation is marked with the label representing the relation name R (line 11), while any other object representing a field for the attribute A_j has A_j as a label, and stores its value into the ξ function (line 13). The τ_{DB} translation function (line 16) provides a representation for a whole database DB using τ_R for all the intermediate results (line 20), where each table is now represented as one single object, nesting the contents of all of its tables.

Algorithm II.5 Semistructured (XML) to GSM

```
 1: function $\tau_{XML}(root_{XML}, seed) : GSM$
 2:     $V := \{0\}$
 3:     $\ell \overset{\text{def}}{=}$ new func. $\varnothing \to \wp(M)$;
 4:     $\xi \overset{\text{def}}{=}$ new func. $\varnothing \to \wp(\mathcal{L})$;
 5:     $\phi \overset{\text{def}}{=}$ new func. $\varnothing \to \mathbb{N}$;
 6:     $\tilde{v} = recursive_{XML}(root_{XML}, \varnothing, \ell, \xi, \phi, [1])$
 7:     return $(\tilde{v}, 0, V, \ell, \xi, \phi)$
 8:
 9: procedure $recursive_{XML}(\, element_{XML}, V, \ell, \xi, \phi, list, seed \,)$
10:     $v \overset{\text{def}}{=} (dt(seed, dt(dtl(list), 0)))_0$;        $V := V \cup \{v\}$
11:     if $element_{XML}$.isTag() then
12:         $\ell(v) := \{\, element_{XML}.\texttt{tag} \,\}$
13:         for $\langle key, value \rangle_j \in \texttt{attributes}(element_{XML})$ do
14:             $v_k \overset{\text{def}}{=} (dt(seed, dt(dtl(list), j)))_0$
15:             $V := V \cup \{v_k\}$;        $\phi(v, \text{“Attribute”}) := \phi(v, \text{“Attribute”}) \cup \{v_k\}$
16:             $\ell(v_k) := \{\, key \,\}$
17:             $\xi(v_k) := \{\, value \,\}$
18:         for $child_j \in \texttt{children}(element_{XML})$ do
19:             ▷ $h :: t$ defines a list where $h$ is the head and $t$ is its tail or rest.
20:             $\tilde{v} = recursive_{XML}(child_j, j :: list)$
21:             $V := V \cup \{\tilde{v}\}$;        $\phi(v, \text{“Tag”}) := \phi(v, \text{“Tag”}) \cup \{\tilde{v}\}$
22:     else
23:         $\ell(v) := \{\, \text{“Text”} \,\}$
24:         $\xi(v) := \{\, element_{XML}.\texttt{getText()} \,\}$
25:     return $v$
```

XML to GSM

Similarly to relational databases, semistructured models have no clear distinction between entities and relationships in their characterization. Moreover, such representations may represent graph entities and relationships using different schemas (see Section 5.4.2 on page 143). As an example for semistructured data, we use the XML model. Please note that a JSON document can be trivially transformed into an XML and hence they may share the same τ_{XML} operator. Similar considerations can be formulated for the nested relational model, or document oriented. We translate XML documents into a GSM, which object containments represent the document's root. We leave to the user the definition of domain specific functions performing a proper translation according to the data's schema. Algorithm II.5 provides the desired transformation: in particular, each **tag node** or **attribute** or **text** is associated to an unique identifier (line 10) and are contained in collections within different properties, where each object is distinguished by its label (respectively lines 12, 16 and 23). Moreover, expression functions ξ are used to store only the associated values for both attributes' values (line 17) and text nodes content (line 24). The indices that will be associated to each nested component will use the usual XML tree indexing function [LZ16] associating to each element a list of identifiers. Such list can then be mapped into one single number (respectively line 21, 15 and again 21) via the *dtl* function (line 14, see Equation 5.4 on page 132).

Algorithm II.6 EPGM to Nested Graph

```
1:  function $\tau_{EPGM}(V, E, L, K, T, A, \tilde{\lambda}, \phi, \omega, \kappa, seed) : N$
2:      $g \overset{\text{def}}{=} dt(seed, dtl(V))_0$
3:      $\phi'(g, \text{"Entity"}) := [dt(seed, dt(i,0))_0 | i \in V \cup L]$
4:      $\phi'(g, \text{"Relationship"}) := [dt(seed, dt(i,o))_0 | i \in E]$
5:      $O := \{\, (dt(seed, dt(i,0)))_0 \mid i \in V \cup E \cup L \,\} \cup \{\, g \,\}$
6:      for each $i \in V \cup E \cup L$ do
7:          $j \overset{\text{def}}{=} (dt(seed, dt(i,0)))_0;$      $\phi'(j) := []$
8:          $\ell(j) := [\kappa(i,\tau)]$
9:          for each $k \in K$ s.t. $\kappa(i,k) \neq \mathsf{NULL}$ do
10:             $h \overset{\text{def}}{=} (dt(seed, dt(i, \mathtt{bin}(k) + 1)))_0;$      $\ell(h) := [k];$      $\xi(h) := [\kappa(i,k)]$
11:             $\phi'(j, \text{"Attribute"}) := \phi'(j, \text{"Attribute"}) \cup [h]$
12:         $O := O \cup \phi(j)$
13:     for each $l \in L$ do
14:         $\phi'((dt(seed, dt(l,0)))_0, \text{"Entity"}) := [(dt(seed, dt(i,0)))_0 | i \in \phi(l)]$
15:         $\phi'((dt(seed, dt(l,0)))_0, \text{"Relationship"}) := [(dt(seed, dt(i,0)))_0 | i \in \omega(l)]$
16:     for each $e \in E$ do
17:         $(s,t) \overset{\text{def}}{=} \lambda(e)$
18:         $\xi((dt(seed, dt(e,0)))_0, \text{"src"}) := [(dt(seed, dt(s,0)))_0]$
19:         $\xi((dt(seed, dt(e,0)))_0, \text{"dst"}) := [(dt(seed, dt(t,0)))_0]$
20:     return $(g, O, \lambda, \ell, \xi, \phi')$
```

EPGM to Nested Graph

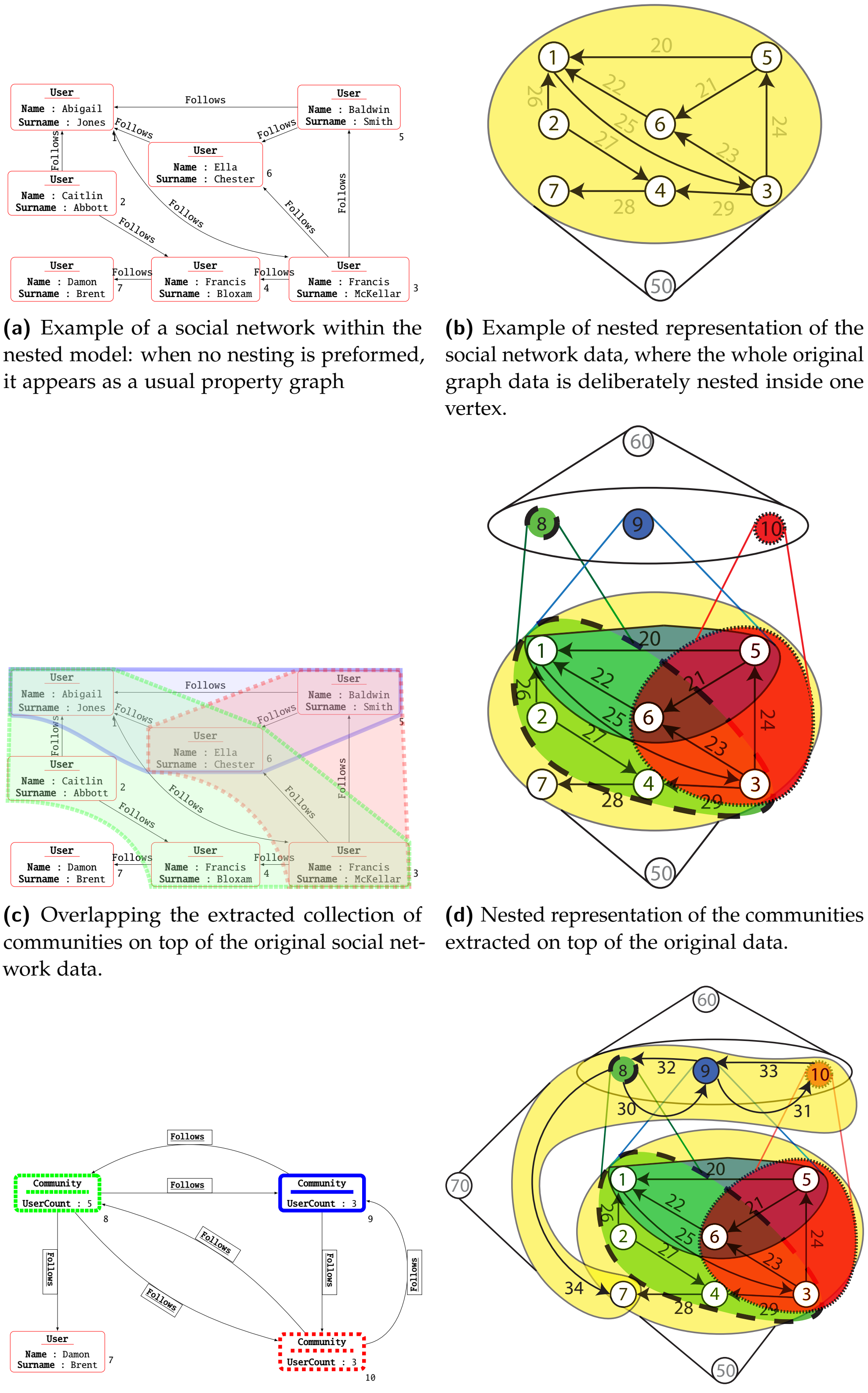
Among all the property graph generalizations, we choose to provide a transformation between EPGM graphs to Nested Graphs because EPGM extends the property graph model with logical graphs. Algorithm II.6 provides the desired transformation to nested graphs. Each vertex is transformed into an "Entity"object (line 3) and each edge into a "Relationship"one (line 4). In order to overcome the limitations of the EPGM model, we decide to represent each logical graph as an "Entity"containing all the "Entity"(ies) in ϕ (line 14) and "Relationship"-s in ω (line 15) as nested components φ. Even in this case, each property value association κ (except from the labels τ) for vertices, edges and logical graphs (line 6) are mapped as nested objects which are neither "Entity"(ies) nor "Relationship"-s (line 9).

5.4 Use Cases

This section will show how our proposed graph data structure can be used in different context and scenarios. We're going to show how nested graphs can be used to represent *part-of* aggregations (Subsection 5.4.1) and how such data structures can be usefully used during the alignment operations (Subsection 5.4.2 on page 143). The usage of is-a aggregations requires to explicitly use some query languages operators, and hence this other aggregation is going to be discussed in Section 6.3.3 on page 182.

5.4.1 Representing *part-of* aggregations

In the following example, we will focus on a social network use cases: we will see how a graph data representation within this model can express both aggregation that are specific to both semi-structured and relational nested models, and graphs.



(a) Example of a social network within the nested model: when no nesting is preformed, it appears as a usual property graph

(b) Example of nested representation of the social network data, where the whole original graph data is deliberately nested inside one vertex.

(c) Overlapping the extracted collection of communities on top of the original social network data.

(d) Nested representation of the communities extracted on top of the original data.

(e) Aggregated representation of the communities, aggregated by the COUNT function over the vertices.

(f) Nested representation of the nested communities on top of the data graph.

Figure 5.5 An example of part-of aggregation within a Social Network. As we can see from the representation, property graphs cannot express the nested components. In order to ease the representation of nested graphs, edges are depicted as arcs instead of objects.

▸ **Example 26.** *Suppose to have asocial network graph containing millions of users, thus making impossible to visually represent the interactions happening within our data. An aggregation of this data helps us to reduce the amount of informations, and hence makes us better understand the connections within the graph. Figure 5.5a represents the initial non-aggregated nested graph: given that it does not contain any nested component, it appears as an usual property graph. This graph can be expressed by our data model sketched in Figure 5.5b as follows:*

$$SN = (50, \{1,\dots,7,20,\dots,29,50,101,\dots,107,111,\dots,117\}, \ell, \lambda, \xi, \phi)$$

$$\phi(50, \text{“Entity”}) = [1,\dots,7] \qquad \phi(50, \text{“Relationship”}) = [20,\dots,29]$$

$$\forall 1 \leq v \leq 7.\ell(v) = [\texttt{User}]$$

$$\phi(1, \text{“Attribute”}) = [101, 111]$$
$$\ell(101) = [\texttt{Name}] \quad \xi(101) = [\text{“Abigail”}] \quad \ell(111) = [\texttt{Surname}] \quad \xi(111) = [\text{“Jones”}]$$

$$\phi(2, \text{“Attribute”}) = [102, 112]$$
$$\ell(102) = [\texttt{Name}] \quad \xi(102) = [\text{“Caitlin”}] \quad \ell(112) = [\texttt{Surname}] \quad \xi(112) = [\text{“Abbott”}]$$

$$\phi(3, \text{“Attribute”}) = [103, 113]$$
$$\ell(103) = [\texttt{Name}] \quad \xi(103) = [\text{“Francis”}] \quad \ell(113) = [\texttt{Surname}] \quad \xi(113) = [\text{“McKellar”}]$$

$$\phi(4, \text{“Attribute”}) = [104, 114]$$
$$\ell(104) = [\texttt{Name}] \quad \xi(104) = [\text{“Francis”}] \quad \ell(114) = [\texttt{Surname}] \quad \xi(114) = [\text{“Bloxam”}]$$

$$\phi(5, \text{“Attribute”}) = [105, 115]$$
$$\ell(105) = [\texttt{Name}] \quad \xi(105) = [\text{“Baldwin”}] \quad \ell(115) = [\texttt{Surname}] \quad \xi(115) = [\text{“Smith”}]$$

$$\phi(6, \text{“Attribute”}) = [106, 116]$$
$$\ell(106) = [\texttt{Name}] \quad \xi(106) = [\text{“Ella”}] \quad \ell(116) = [\texttt{Surname}] \quad \xi(116) = [\text{“Chester”}]$$

$$\phi(7, \text{“Attribute”}) = [107, 117]$$
$$\ell(107) = [\texttt{Name}] \quad \xi(107) = [\text{“Damon”}] \quad \ell(117) = [\texttt{Surname}] \quad \xi(117) = [\text{“Brent”}]$$

$$\forall 20 \leq e \leq 29.\ell(e) = [\texttt{Follows}]$$

$$\begin{array}{llll}
\phi(20, \text{“src”}) = [5] & \phi(20, \text{“dst”}) = [1] & \phi(21, \text{“src”}) = [5] & \phi(21, \text{“dst”}) = [6] \\
\phi(22, \text{“src”}) = [6] & \phi(22, \text{“dst”}) = [1] & \phi(23, \text{“src”}) = [3] & \phi(23, \text{“dst”}) = [6] \\
\phi(24, \text{“src”}) = [3] & \phi(24, \text{“dst”}) = [5] & \phi(25, \text{“src”}) = [1] & \phi(25, \text{“dst”}) = [3] \\
\phi(26, \text{“src”}) = [2] & \phi(26, \text{“dst”}) = [1] & \phi(27, \text{“src”}) = [2] & \phi(27, \text{“dst”}) = [4] \\
\phi(28, \text{“src”}) = [4] & \phi(28, \text{“dst”}) = [7] & \phi(29, \text{“src”}) = [3] & \phi(29, \text{“dst”}) = [4]
\end{array}$$

Please note that vertex 50 *actually nests a full representation of a graph and consequently, we have that each vertex or edge may nest a whole graph within* ϕ.

By using community detection algorithms, we extract a set of graph collections in polynomial time with respect to the data size [vDAG12], where some overlaps between communities may be present. Figure 5.5c presents such extracted communities as shaded areas on top of the original data sources: please note that the property graph model does not allow to represent such communities inside the same given graph. This problem can be overcome by our proposed data structure as showed in Figure 5.5d on page 140: each community, marked with a different colour, is nested inside one given object. In particular we have that:

$$\phi(8,\text{“Entity”}) = [1,2,3,4,6] \quad \phi(8,\text{“Relationship”}) = [22,23,25,26,27,29]$$

$$\phi(9,\text{“Entity”}) = [1,5,6] \quad \phi(9,\text{“Relationship”}) = [20,21,22]$$

$$\phi(10,\text{“Entity”}) = [3,5,6] \quad \phi(10,\text{“Relationship”}) = [21,23,24]$$

As we can see from the same picture, we can create a graph collection by creating another object including the three objects nesting the communities:

$$\ell(60) = \{\,\texttt{Communities'Collection}\,\} \qquad \phi(60,\text{“Entity”}) = [8,9,10]$$

The aggregation function evaluating how many users are contained within the community can be expressed with a `script` *expression as follows:*

$$\ell(8) = \ell(9) = \ell(10) = \{\texttt{Community},\texttt{UserCount}\}$$

$$\xi(8) = \xi(9) = \xi(10) = [\text{“}\texttt{0+(o.phi ["Entity"])}\text{”}]$$

Please observe that each expression in $\xi(o)$ is going to be evaluated by choosing “`o`*” as o.*

As a result, we would like to summarise each component as a single vertex containing all the vertices and edges describing the communities, as represented by Figure 5.5e, where only the result of the aggregation is provided. Given the outline provided by the former example, it is easy to define a preliminary algorithm that takes both the outcome of the community detection algorithm and the social network graph, and aggregates each community as a single vertex: ***(a)*** *given a graph with its vertex set V, return as V' the vertices that do not appear within any detected community,* ***(b)*** *alongside with the aggregated representation of each community. As a result of this vertex creation phase, vertices 8, 9 and 10 are selected alongside with 7, which does not belong to any community. As a result we have that:*

$$\phi(70,\text{“Entity”}) = [7,8,9,10]$$

In a later step, we must also define the criteria by which we have to generate the edges among the V' returned vertices. In particular, we must consider that new edges may occur between vertices where: ***(i)*** *source comes from phase* ***(a)*** *and targets from* ***(b)*** *(or vice versa),* ***(ii)*** *or if vertices from phase* ***(a)*** *are linked to at least one vertex appear inside an aggregated component from* ***(b)*** *(or vice versa),* ***(iii)*** *or even if one vertex from phase* ***(b)*** *contains a vertex that is linked to another vertex represented inside an aggregated component from* ***(b)****. If we want to simply establish a link for all the aforementioned cases, then the result can be modelled within our nested graph definition.*

The result of such aggregation is depicted by the nested element 70, containing both the vertices and the edges extracted following the previous sketched algorithm.

$$\phi(70,\text{“Relationship”}) = [30,31,32,33,34]$$

Given that edges can also contain graph nested components, we could also decide that each edge obtained after the vertex aggregation phase can show how many users (vertices) are shared between the two communities and provide which vertices and edges from the source and target communities allowed the creation of the new link. As we can observe from this latter description, we have that the graph collections required to perform the nesting can be returned by an external algorithm, while the link creation process can be derived from a previous pattern matching process on top of the previously vertex aggregated data. On the other hand, since current graph traversal and graph pattern matching languages do not formally support nested graphs, it is impossible to use currently implemented languages to express the vertex containment. On the next chapter we will then discuss on the features that are required by the graph query languages on top of this novel data structure.

5.4.2 Graph ETL and $\varrho^{\varsigma(-)}_{\alpha(D_i),H}(\alpha(D_i))$: the Transformation phase

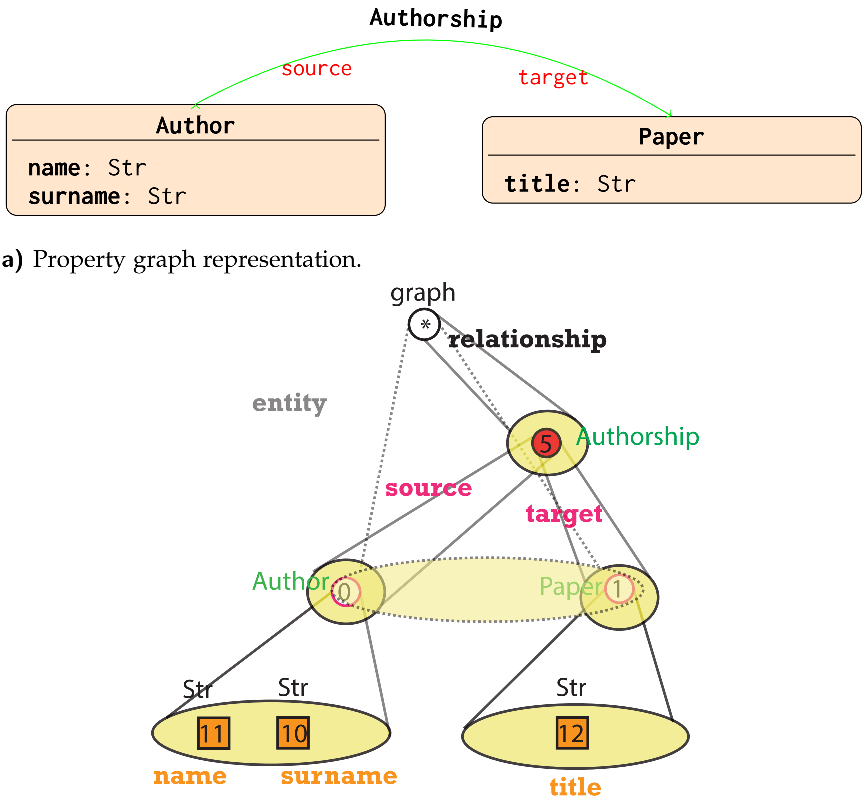
Despite the attempts of defining graph data warehouses [EV12a, EV12c, ZLXH11], a complete outline of graph ETLs is still missing, that is the process of integrating multiple heterogeneous graph data sources into one final graph [CYZ⁺08]. All the required components for such process are already described in literature, and consists into two main phases, called *transformation* and *loading*. From the following definition it will be also evident that usually ETLs use a different sequence of operations than the one prospected by the previous GAV approach. Nevertheless, we're going to walk in the footsteps of GAV, as it provides a better formal approach.

Concerning the *transformation* phase, we first have to transform our data into a common representation. After doing so, we can clean data ($\nu_{\cong}$) and resolve all the ambiguities [Rah16]. Then, we consider the informative need of the user expressed through a hub schema (called "conceptual graph model" in Data Warehouse contexts [JFL15]), expressing the graph schema to which the final graph must be compliant with. This graph schema, expressed as a graph pattern query, is used for aligning each graph data source [AGG⁺15] over a same representation through either approximate pattern matching [DVMT15] or the aforementioned ϱ operator. After the alignment phase, the data sources are finally transformed to match the alignment schema.

In the *loading* phase, the graphs resulting from the previous phase are integrated into one final graph. As a consequence, the generalization of graph joins with disjunctive semantics in **full graph joins** is required. Nevertheless, graph joins are not flexible enough within a general data integration scanario. We will continue such discussion in Section 7.1.1 on page 199.

This section will focus on one possible definition of the ϱ operator, ϱ $\varrho^{\varsigma(-)}_{\alpha(D_i),H}(\alpha(D_i))$, within the data transformation phase embedding an alignment transcoding step ς: by doing so we show that GSM provides a common representation for comparing semistructured data in JSON and graphs through the definition of correspondences, and describe how the result of the alignment can be differently interpreted with respect to the different information need. Another definition of such operator within our proposed query language (defined as a generalization of the graph grammar approach) is going to be described on the next chapter at Section 6.3.4 on page 183.

After introducing the graphs as data structures on which we mined the correspondences for the alignment, we now want to generalize the data integration approach for semistructured data and choose to integrate them into graphs. This time we will not integrate semistructured data using merged schema between the two sources as in Section 2.1.2 on page 21, but we will use a graph of choice as a target schema, thus resembling more what it has been outlined for ϱ in the LAV/GAV scenario.



(a) Property graph representation.

(b) Nested graph representation.

Figure 5.6 Representation of the hub schema providing the final desired representation of the JSON files after the alignment phase.

In order to simplify the more general problem, we will use as input JSON (semistructured) data representing graphs using two different kinds of schemas. We use some notation that was already introduced during the formalization of schema correspondences and data modelling (Section 2.2.1 on page 34). In particular, we will start to analyse the data alignment applied to data provided with different non-GSM representation, both as inputs and outputs.

▸ **Example 27.** *Suppose to have two bibliographic networks represented as bipartite graphs in JSON files using different schemas (see Example 7 on page 41): we now want to integrate them into one final single graph. Such formats are provided in Figure 5.7 (g for short) and Figure 5.8 (g' for short). Their respective schemas $\alpha(g)$ and $\alpha(g')$ are provided in Figure 5.9, using the same syntax adopted in the previous introductory examples. As in the Data Warehouse scenario, the querying user provides the hub schema [GHKR11, HGR13] in Figure 5.6a as a property graph: such schema already provides which fields shall be contained by the vertices, and which are the labels to be associated to authors, papers and edges connecting them. All the possible data sources must be mapped to this hub schema and then aligned towards it [ES13]. Thus correspondences will tell how to transform the data source instance into the global schema definition, thus completing the data integration task.*

Now we should consider if the schema extraction α from data sources jointly with the usual data structures represented in a non uniform representation (JSON data and graph schema), can be

```
{ "graph": {
      "nodes": [{
          "id": "0", "label": "Author",
          "metadata": { "name": "Abigail", "surname": "Conner" }
      },{
          "id": "1", "label": "Author",
          "metadata": { "name": "Cassie", "surname": "Norman" }
      },{
          "id": "2", "label": "Author",
          "metadata": { "name": "Baldwin", "surname": "Oliver" }
      },{
          "id": "3", "label": "Paper",
          "metadata": { "title": "On␣Joining␣Graphs" }
      },{
          "id": "4", "label": "Paper",
          "metadata": { "title": "Object␣Databases" }
      },{
          "id": "5", "label": "Paper",
          "metadata": { "title": "On␣Nesting␣Graphs" }
      }],
      "edges": [{
          "id": "6", "label": "AuthorOf",
          "source": "0", "target": "3",
          "metadata": {}
      },{
          "id": "7", "label": "AuthorOf",
          "source": "1", "target": "3",
          "metadata": {}
      },{
          "id": "8", "label": "AuthorOf",
          "source": "1", "target": "4",
          "metadata": {}
      },{
          "id": "9", "label": "AuthorOf",
          "source": "2", "target": "4",
          "metadata": {}
      },{
          "id": "10", "label": "AuthorOf",
          "source": "2", "target": "5",
          "metadata": {}
      }]
    }
}
```

Figure 5.7 JSON representation g of the graph presented in Figure 2.17a using the JSON Graph Format. `http://jsongraphformat.info/`

```
[
  { "id":"0",
    "data":{ "name": "Abigail", "surname": "Conner" },
    "meta":{"label":"Author","graphs":["11"]}
  },{
    "id":"1",
    "data":{ "name": "Cassie", "surname": "Norman" },
    "meta":{"label":"Author","graphs":["11"]}
  },{
     "id":"2",
     "data":{ "name": "Baldwin", "surname": "Oliver" },
     "meta":{"label":"Author","graphs":["11"]}
  },{
    "id":"3",
    "data":{ "title": "On␣Joining␣Graphs" },
    "meta":{"label":"Paper","graphs":["11"]}
  },{
    "id":"4",
    "data": { "title": "Object␣Databases" },
    "meta":{"label":"Paper","graphs":["11"]}
  },{
    "id":"5",
    "data":{ "title": "On␣Nesting␣Graphs" },
    "meta":{"label":"Paper","graphs":["11"]}
  }
]
```

(a) Vertex representation, v'. Instead of representing both field "name" and "surname, in this case we provide a "fullname" field for data integration explanatory reasons.

```
[
  { "id":"6", "source":"0", "destination":"3",
    "data":{},
    "meta":{"label":"AuthorOf","graphs":["11"]}
  },{
    "id":"7", "source":"1", "destination":"3",
    "data":{},
    "meta":{"label":"AuthorOf","graphs":["11"]}
  },{
     "id":"8", "source":"1", "destination":"4",
     "data":{},
     "meta":{"label":"AuthorOf","graphs":["11"]}
  },{
    "id":"9","source":"2","destination":"4",
    "data":{},
    "meta":{"label":"AuthorOf","graphs":["11"]}
  },{
    "id":"10","source":"2","destination":"3",
    "data":{},
    "meta":{"label":"AuthorOf","graphs":["11"]}
  }
]
```

(b) Edge representation, e'. The original tag "target" is here represented with the name "destination" for data integration explanatory reasons.

Figure 5.8 JSON representation $g' = (v', e')$ of the graph presented in Figure 2.17a using the GRADOOP JSON format `http://gradoop.org/`, where vertices and edes are represented in different files.

```
{ graph: { nodes: [({ id: Str,
                      label: Str,
                      metadata: { name: Str, surname: Str} +
                                { title: Str }
                            })*],
                  edges: [({ id: Str,
                             label: Str,
                             source: Str,
                             target: Str,
                             metadata: {}
                           })*],
                }
}
```

(a) Schema $\alpha(g)$ associated to the g representation in Figure 5.7

```
[{ id: Str,
   data: { fullname: Str} + { title: Str},
   meta: { label: Str,
           graphs: [Str*]
                 }
}*]
```

(b) Schema $\alpha(v')$ associated to the v' representation in Figure 5.8a

```
[{ id: Str,
   data: {},
   source: Str,
   destination: Str,
   meta: { label: Str,
           graphs: [Str*]
                 }
 }*]
```

(c) Schema $\alpha(e')$ associated to the e' representation in Figure 5.8b

■ **Figure 5.9** Extracting the schema associated to the JSON files representing some graphs. The same syntax of [BLC+17] is adopted.

supported by one single DBMS with one single query language. This example will show that the proposed data models are not able to draw correspondences between some internal representation components, such as single values or attributes. Since the correspondence of the Authorship edge depends on the prior correspondence of the entities Author and Paper, we must resolve the latter first, and the former in a subsequent step. For the moment, let us focus on the correspondences between $\alpha(g)$ and the hub schema, with respect to the nodes' information: $\alpha(g)$ tells us that the field metadata contains either the authors' information (name and surname appear in one option) or the paper's information (title appears as the other alternative). Even if this solution could be considered valid for distinguishing authors from papers papers within the data source (those elements are fully described by such disjunct set of attributes), in some real cases the dependency between the label and the contained attribute-value association could be helpful to distinguish the most relevant correspondence [oRD12]. Since the information of such labels is lost in the schema extraction process because the label is expressed as a value, we must extend the schema with some inference rules, thus allowing to define a complete ontology describing the data:

```
if (graph.nodes.metadata : {name: Str, surname: Str}) then
        (label : Str) = "Author"
else
        (label : Str) = "Paper"
```

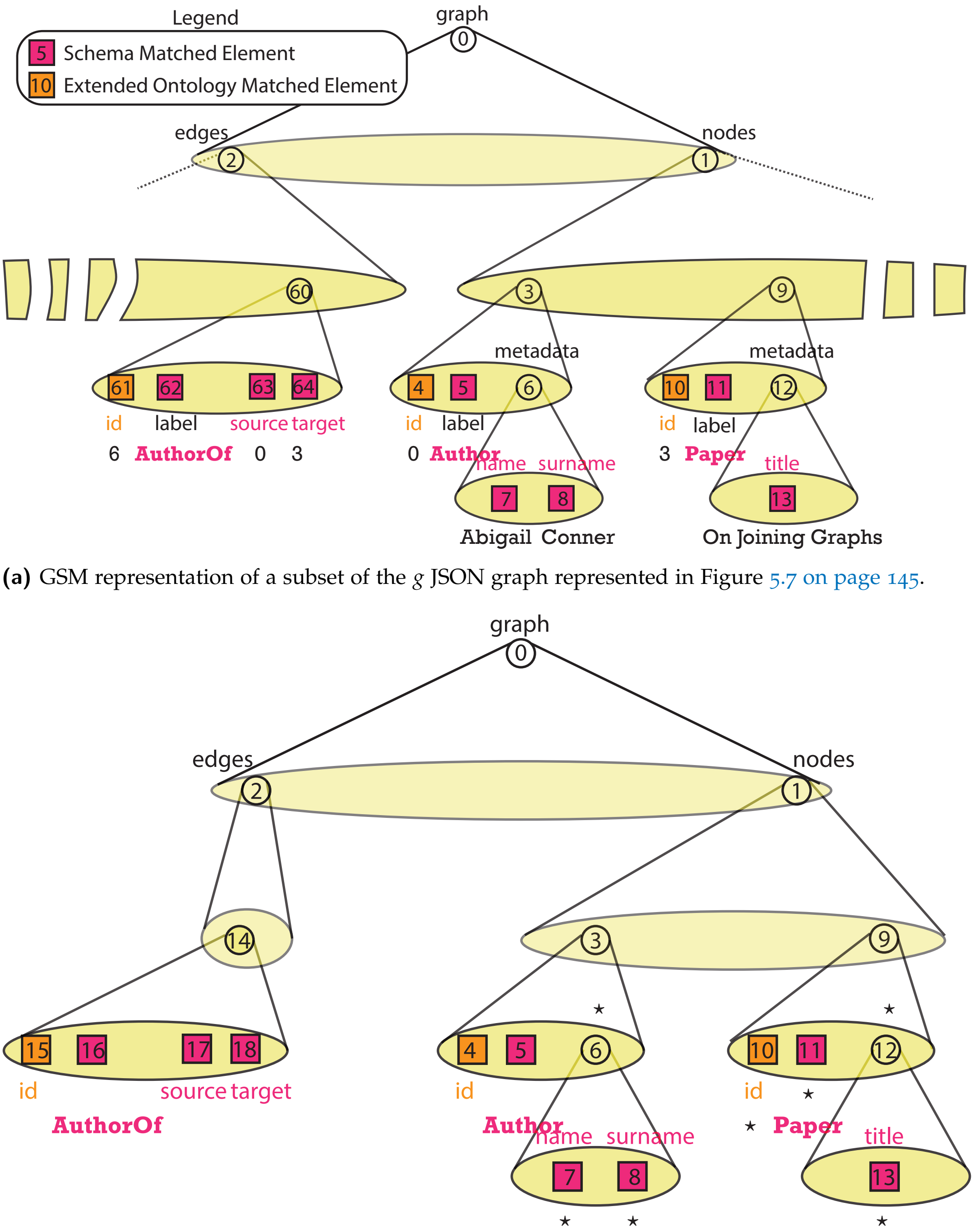
Such rules could be extracted using associative rules [TSK05] but can be represented by neither the schemas nor the data sources. This extension is although required because it is also impossible to draw correspondences as edges between the hub schema's attributes (belonging to either Author or Paper) and the fields of the JSON format. This is also true because, in traditional property graphs, attributes cannot be connected to other components via edges. Given that edges may contain other properties, this problem will be also be present in the edge alignment phase.

Let us move on and discuss the relationship alignment phase. The hub ontology already tells that an Authorship relation associates an Author (source) to a Paper (target). We now want to match such definition for the edges contained in the JSON representation g: even in this case an edge has a field called source and target as in the hub schema, but within our JSON representation source and target do not explicitly syntactically refer to a vertex, because source and target information is represented as a string (Str). Even if we stopped the analysis at this point, we would notice that it is impossible to draw scuh correspondences as an "edge" between the edge's source within the hub schema and any object represented in $\alpha(g)$. Even in this case, we must represent correspondences at a metalevel because no direct data manipulation is possible in practice.

Let us continue to draw correspondences at the meta level. Our data integration system already knows that Author and Paper in the hub schema are "Entity"(ies), and that Authorship is an edge connecting a vertex of the first type to a vertex of the latter. We can also freely assume that an internal dictionary will tell the system that target and destination are synonyms and that they both refer to edges. Similar considerations could be done for $\alpha(g)$, where only the edges field will be detected as containing a collection of edges. At this point, a word ontology such as Babelnet [NP12a] could tell that the term Authorship is related to the term AuthorOf used in g representing an edge. Moreover, such JSON edges have a field called source and target, and hence there should be a correspondence between the JSON AuthorOf and the final graph Authorship edges, e.g. remarked by an i variable:

```
graph.edges[i] : {source: Str, target: Str, label: Str, ...}
graph.edges[i].label = "AuthorOf"
```

Moreover, with the previous alignment phase we also have found that Author and Paper are nodes, and then there should also be a correspondence between source and target as follows:



(a) GSM representation of a subset of the *g* JSON graph represented in Figure 5.7 on page 145.

(b) Schema extracted from the aggregation over the schema components found in the former GSM representation. Stars represent placeholders for the underlying data that has been ignore in the schema generation phase.

Figure 5.10 Manipulation of the JSON data for schema extraction.

```
graph.nodes[source] : {label: Str, ...}
graph.nodes[source].label = "Author"

graph.nodes[target] : {label: Str, ...}
graph.nodes[target].label = "Paper"
```

In order to close the correspondence diagram, now we only have to detect which is the correspondence within the hierarchy associating each edge to a vertex: this could be done as previously through the extraction of associative rules, and hence we could determine that source and target within g must refer to the nodes' id-s, thus allowing to extend the general edge rule as follows:

```
graph.edges[i] : {source: Str, target: Str, label: Str, ...}
graph.edges[i].label = "AuthorOf"
graph.edges[i].source = source
graph.edges[i].target = target
```

We completed the alignment process via the enrichment of the hub schema through the extraction of associative rules. At this stage the alignment of g with the hub schema is completed, and then we can use such correspondences to perform the sources' translation towards the hub schema representation. ◂

After describing how to carry out the data alignment within these two different data models (JSON and property graphs), we want to show how the previous data alignment process can be defined without any additional meta-level information by using GSM, which allows to represent both alignments and schemas. While in the previous example we needed to express further predicates and association rules in a distinct layer different from the data representation, in this incoming example such correspondences (morphisms) are going to be expressed through an edge representation as introduced in Definition 18 on page 77. If we want to create (hyper)edges having as targets the objects o in the pattern P and as sources one of the objects $\bigcup_{f_i \in m_P(\eta)} f_i(o)$ in η, such edges can be represented via the following object set:

$$w_{m,P,\eta}(o) = \{\, e_{oo'} \mid f_i \in m_P(\eta) \Rightarrow \phi(e_{oo'}, \text{“src”}) = f(o) \wedge \phi(e_{oo'}, \text{“dst”}) = [o] \,\}$$

On the other hand, the function providing all the edges from the η data source towards the matched object in P is defined as follows:

$$\text{ʍ}_{m,P,\eta}(o') = \left\{\, e_{oo'} \,\middle|\, f_i \in m_P(\eta), o' \in f_i(o) \Rightarrow \phi(e_{oo'}, \text{“src”}) = [o'] \wedge \phi(e_{oo'}, \text{“dst”}) = [o] \,\right\}$$

In the next example we are going to focus on how such data structures can support correspondences' transcodings, while Chapter 7 will show how to instantiate and transform the matched parts after the definition of a query language over such data structures. Appendix A on page 225 provides the full source code describing the following example.

▸ **Example 27** (continuing from p. 144). *We now must translate the hub schema into a nested graph representation as in Figure 5.6b on page 144. This other representation explicitly tells us that* “Relationship”*s have structural dependencies on* “Entity”*(ies), because each edge contains the information for the vertices. Hereby, the implication "entities must be reconstructed before the edges" comes for free from the traversal of the hub schema itself. Therefore, the schema alignment can be carried out by extracting the data's schema from the GSM-translated representation of the JSON objects: in particular, we can first perform a linear scan of the data (Figure 5.10a on the preceding page) where the key elements of interest of the hub schema are selected (in magenta) alongside with* *other relevant terms (in orange). Then again, we aggregate each object in the data structure by the hub schema elements matched by each object, and hence we obtain the schema in Figure 5.10b. Please note that the outline of the abstraction function α_1 extracting the schema from the data acts similarly to the aggregation operator*[15]*, and then new edge morphisms in $w_{\star,\alpha_1(g),g}$ are generated: in particular, $\star$ represents an arbitrary summarized value that can be induced by the aggregation $\alpha_1(g)$ of the input data (g); moreover, $\alpha_1(g)$ actually represents the schema for g as required in the previous examples. This means that GSM, as well as graphs semistructured data representations, allow to represent* DATA *g and* MODEL *$\alpha_1(g)$ at the same representation level.*

After "aligning" the data object g to the source schema ones $\alpha_1(g)$ obtained through aggregations of the objects in g, we want to continue with the (proper) schema alignment phase between the source schema and the elected hub schema: Figure 5.11a on the next page shows a traditional way to compare schemas, where correspondent entities are matched (even with this case) with an edge. Please also note that edges may be represented as nested graphs' edges and hence as objects even though, for representational ease, we distinguish such objects from the schema ones by representing them as arcs, similarly to what we did in the previous social network scenario. This schema matching representation must distinguish[16] *the correspondences' edges in two main types:* ***(1)*** *orange dashed edges remark "ℓ"-matches that are objects containing similar (or identical) labels, and* ***(2)*** *black uniform edges remark "ξ"-matches establishing correspondences between objects $\tilde{o}$ in the source schema and hub schema objects $\tilde{o}'$, appearing similar "constrainable values" according to a similarity predicate ϑ; in particular the constrainable values are contained by either ξ functions (e.g., $a \in \xi(\tilde{o})$ and $a' \in \xi(\tilde{o}')$) or the attribute through which the object is contained via ϕ (e.g., $a \in \{\, p \mid \exists o'.\tilde{o} \in \phi(o',p) \,\}$ and $a' \in \{\, p' \mid \exists o'.\tilde{o}' \in \phi(o',p') \,\}$). All the black arrows showed in Figure 5.11 on the following page belong to such second scenarios, because the "source" and "target" values appearing in the source schema match with the containment attribute on the hub schema. Therefore, we can define the set of all the ξ-constrainable values appearing at a maximum*[17] *height δ from o as follows:*

$$cv_\delta(o) \overset{\text{def}}{=} \xi(o) \cup \left\{\, a \;\middle|\; \exists o'.o \in \phi(o',a) \,\right\} \cup \bigcup_{o' \in \varphi^\delta(o)} cv(o')$$

over which we can define our desired notion of object similarity over constrainable values[18] *via a similarity predicate ϑ:*

$$cv_{\delta,\vartheta}(o,o') \Leftrightarrow \exists a \in cv_\delta(o).\exists a' \in cv_\delta(o').\vartheta(a,a')$$

While the "ℓ"-matches identify the possible translation of an entity in two different representations by comparing the MODEL *pieces of information, the "ξ"-matches identify the constraints that must be preserved in the transformation phase. Please also note that these considerations are valid if the data of choice represents the objects' types through ℓ, and values or constraints through ξ-s.*

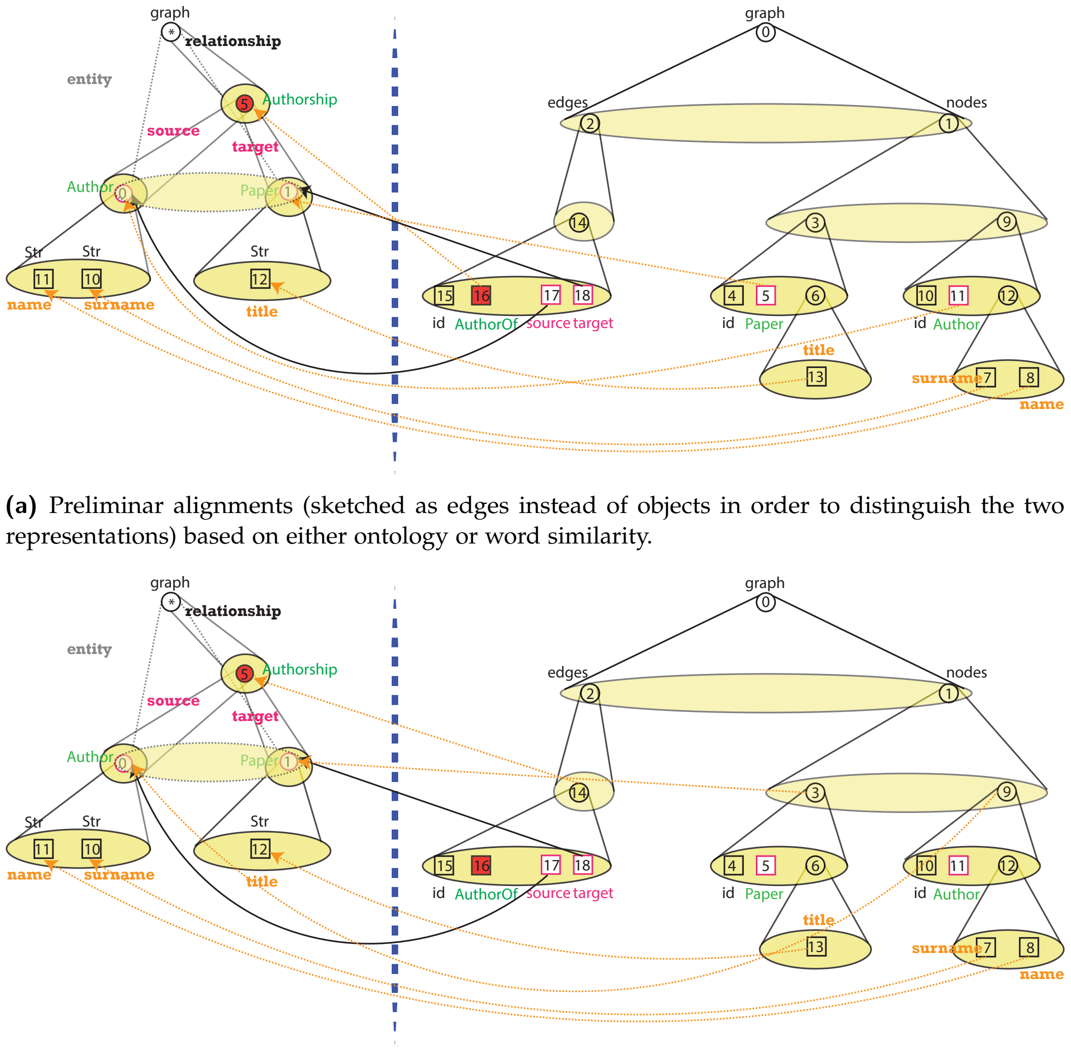
We can now observe that such "raw" correspondences can be refined by transforming them with ◁ Alignment's refinement

[15] This operation will be formalized as a derivate GSQL operator at page 170. We can also think that the final id of the aggregated components depends on the list dovetailing *dtl* of all the elements that have been aggregated, thus easing the subsequent transformation phase by reducing the visit time of the data data structure.

[16] Such distinction, which is self-evident during the match creation phase, can be saved (e.g.) by storing the correspondences in different collections of a same object ω; as an example $\phi(\omega, \text{"ell"})$ and $\phi(\omega, \text{"xi"})$ containments may collect these two types of correspondences.

[17] See line 192 at Appendix A.

[18] See line 215 at Appendix A.



(a) Preliminar alignments (sketched as edges instead of objects in order to distinguish the two representations) based on either ontology or word similarity.

(b) Refactoring the previous alignments by comparing the depth of where such elements are located.

Figure 5.11 Depicting two distinct phases of the entity alignments. Both the ξ (dashed and in orange) and ℓ alignments (in a black continuous line) are represented as object in order to visually distinguish them from the source (on the right) and hub schema (on the left) objects.

a ς: while in the hub schema the `Author` *node contains the attributes* `name` *and* `surname`*, in the data schema we have that the latter information is contained in a sibling object; therefore, the previous correspondences need to be post-processed to match each element of the hub schema with the coarsest representation within the data model. Therefore, we must reconcile each ℓ-correspondence to the nearest ξ-correspondence, so that the reconciliation of the schema corresponding objects (happening on the next phase) will be eased. This ς process can be carried out with a depth postVisit*[19] *of the hub schema starting from its reference object, where the following operations are performed over the object o:*

[19] See line 469 at Appendix A.

- *If the current element o represents a leaf, no transformation occurs.*
- *Otherwise, after performing the preVisit*[20] *on all the contents in $\varphi(o)$, update the correspondences having o as a target, so that they are mapped to the $\alpha_1(g)$ coarser representation describing o and $\varphi(o)$. This post condition is satisfied by the following procedure:*

  Given T the set of the ℓ-matched having their targets $\{o\} \cup \phi(o)$ in the hub schema, let S be the set of their sources in the source schema ($S \overset{\text{def}}{=} \bigcup_{t \in T} \phi(t, \text{"src"})$), we want to select the set of strongly nested object-set with respect to $\varphi^*(S)$, that is $\varphi^*(S)$ itself by definition. Now, we group all the correspondences in T by their sources' membership to a specific element $a \in \varphi^*(S)$, that is:

  $$T_S \overset{\text{def}}{=} \{\, \{\, t \in T \mid a \in \phi(t, \text{"src"}) \,\} \mid a \in \varphi^*(S) \,\}$$

  From each non empty collection $c \in T_S$, choose just one ℓ-match $t_c \in c$ having a target object $o^{t_c}_{dst}$ (e.g., $o^{t_c}_{dst} \in \phi(t_c, \text{"src"})$) maximizing the absolute relative height rh over $o' \in \{o\} \cup \varphi(o)$:

  $$t_c \overset{\text{def}}{=} \arg\max_{\substack{t \in c, \\ t \in \phi(\omega, \text{"ell"})}} \left\{\, rh(o', \tilde{o}) \,\middle|\, \tilde{o} \in \phi(t, \text{"dst"}), o' \in \{o\} \cup \varphi(o) \,\right\}$$

  Last, update t_c's source to the common ancestor(s) of both ℓ and ξ matches originating (i.e., having their sources) in $\phi(a_c)$, where $a_c \in \phi(t_c, \text{"src"})$.

Figure 5.11b depitcs the outcome of such process: while the matches over leaf objects subsume no refactoring, the others are refactored; for example, the `Author` *element in the hub schema perfectly matches with the author representation within the* `nodes` *collections. Similar considerations can be carried out for the edges.*

After consolidating the ℓ-correspondences, we want to resolve both the morphisms and the alignment transformation by associating the data objects to the hub schema. Please note that the association of the results is only a preliminary step leading to the schema instantiation into some actual data. Such final process is going to be discussed in more detail in the next chapter with the help of GSQL, an ad hoc query language for GSM data representations.

◁ Associating the data to the hub schema

At this point, we must ask ourselves how to traverse both schemas, hub and source, in order to associate the data objects to the final representation (similarly to the Pop illustrated in Section 3.1.1.1 on page 58): this can be carried out by both traversing the schemas' alignment and the morphisms linking the data to its schema representation. This operation can be carried out using a postVisit [21] *over the hub schema objects, starting from the reference object.*

For each currently visited object ω, postVisit[22] *the contained objects $\varphi(\omega)$ first; the contents are accessed by ordering them from the highest one with respect to the definition of h_o. For each visited object $o_i \in \varphi(\omega)$, the complete list of associations $\mathcal{I}_{o_i}$ is going to be returned*[23]*; each list $\mathcal{I}_{o_i}$ has an associated schema $\mathcal{S}_{o_i}$, which is a list of objects in the source schema to which o_i corresponds via ℓ-correspondences and subsequent refinements ($\mathcal{S}_{o_i} = \{\, \iota_{i_1}, \dots, \iota_{i_m} \,\}$); consequently, each element $a \in \mathcal{I}_{o_i}$ is a list of lists $a = \left\{\, \lambda^a_{i_1}, \dots, \lambda^a_{i_m} \,\right\}$ where $\lambda^a_{i_j}$ is a list of data objects corresponding to an object ι_{i_j} via the morphism*[24] *$\iota_{i_j} \to \lambda^a_{i_j}$. In order to use such $\lambda^a_{i_j}$ as constraints for the objects associated to our current object, we must first extract the set $\mathcal{I}_\omega$ of all the objects associated to it. This procedure*[25] *is defined as follows:*

[20] This means that o is evaluated by the visiting algorithm before any other of its contents $\varphi(o)$.

[21] This means that, before visiting o, we postVisit the contents $o' \in \varphi(o)$, from the highest o' to the lowest o' w.r.t. $h_o(o')$, and then we evaluate o' as a last step.

[22] See line 492 on page 231 at Appendix A.

[23] See line 475 on page 231 at Appendix A.

[24] Refer to Definition 18 on page 77 for this notion of morphism.

[25] See line line 398 on page 230 at Appendix A.

> *If **(i)** ω is a target for no ℓ-correspondences, then $\mathcal{S}_\omega$ is the join of all the sources' schemas ($\mathcal{S}_\omega = \bowtie_{i \leq n} \mathcal{S}_{o_i}$) and $\mathcal{I}_\omega$ is the join between all the contents' associated values ($\mathcal{I}_\omega = \bowtie_{i \leq n} \mathcal{I}_{o_i}$). Otherwise, **(ii)** $\mathcal{S}_\omega$ is the set $\left\{ \jmath_{\omega_1}, \dots, \jmath_{\omega_m} \right\}$ of all the source schema objects $\jmath_{\omega_i}$ linked to ω via a ℓ-correspondence $\jmath_{\omega_i} \to \omega$ and all the elements $b \in \mathcal{I}_\omega$ are lists of lists $b = \left\{ \lambda^b_{\omega_1}, \dots, \lambda^b_{\omega_m} \right\}$ where $\lambda^b_{\omega_i}$ is a set of data objects, which is associated to the source schema $\jmath_{\omega_i}$ via a morphism $\jmath_{\omega_i} \to \lambda^b_{\omega_i}$. This means that, in the second case, the data object referring to the parent ω are not (necessarily) restricted via the containment information. The only refinements may only occur via ξ correspondences, as described in the following paragraph.*
>
> *Next, if **(iii)** ω is a target for no ξ-correspondences, $\mathcal{I}_\omega$ is not restricted (or filtered). Otherwise, **(iv)** we must first create a set $\mathcal{I}_{\xi,\omega}$ for such correspondences, constructed similarly to $\mathcal{I}_\omega$ for the ℓ-correspondences, and hence having an associated schema $\mathcal{S}_{\xi,\omega}$; then, for each $b \in \mathcal{I}_\omega$, we preserve each set $\lambda^b_{\omega_i}$ where there is at least one element satisfying one of the values associated to the objects in e, for each $e \in s$, where s is contained in $\in \mathcal{I}_{\xi,\omega}$.*

*After defining $\mathcal{I}_\omega$, we can further skim the $\mathcal{I}_\omega$ using $\bowtie_{i \leq n} \mathcal{I}_{o_i}$ over $\mathcal{I}_\omega$, similarly to the steps outlined in **(iv)**[26]. After this further filtering, we can return $\mathcal{I}_\omega$ to all the possible containers, and hence terminate the postVisit for ω. After terminating the postVisit for the reference object of the hub schema, we can use the instantiated hub schema and either return a graph by unnesting all the matched elements as it will be remarked for the Pop operator presented at page 177 or even transform the content, as it will be showed in Section 6.3.4 on page 183.*

Please observe that the alignment of g' with the hub schema could be carried out similarly, except for the process requiring to align `fullname` *with both* `name` *and* `surname`. *If both g and g' share a common subset of data sources, then we could try to perform a match between all the values associated to the attributes defining an* `Author` *[LJ14]. Since in $\alpha(v')$ there is just one field for the* `Authors`, `fullname`, *while in $\alpha(g)$ we have both* `name` *and* `surname`, *we could try to find some correspondences and manage to find that* `fullname` *is just a combination of* `name` *and* `surname`. *When such task is not generally possible because the two bibliographic network belong to different conferences with authors with different research interests, then the only way to associate* `fullname` *with* `name` *and* `surname` *is to use an ad-hoc ontology containing informations regarding such ontology, thus allowing to perform the whole alignment process.*

This last example showed us that the whole alignment process of the hub schema towards the dataset via the dataset's schema resembles an approximated alignment. In particular such matches are refined through a ς transcoding based on the data content through the correspondences. After this preprocessing and data association phase, we can manipulate the matched data by associating it to the hub schema through φ containment relations. In the next chapter we are going to see a different approach for data integration, where the data matched by traversing the morphisms may be also transformed into new object not pre-existing the data source.

[26] See line 368 on page 229 at Appendix A.

5.5 Conclusions

This chapter allowed to introduce the topic of data integration: as required by the Global as a View scenario, a general data representation for integrating different data sources is provided. This intermediate representation is the Generalized Semistrucured Model, GSM for short, which allowed the definition of the τ translation operators required by Definition 7 on page 52. If we have some more precise information concerning the original data, we can then elect which objects have to be considered as entities and which others are promoted to relationships: after this intermediate step, we can now transform a GSM into a Nested Graph by providing the schema to which the final data must comply to.

Another similar approach for data matching and rewriting is going to be provided in the next chapter as a generalization of the graph grammars.

The last section also showed how GSM promotes a better alignment between entities and attributes (both represented as objects). In particular, the data objects are associated to the hub schema H via the (intermediate data) schemas $\alpha(D)$: after matching the data to its schema and refining the correspondences, we obtain a grammar rule $\alpha(D) \to H$. Then, by filtering out the data in D, we create direct correspondences between D and H. After a data cleaning phase which on which we will not linger because it is a topic already addressed in literature ($\nu_{\cong}$), we'll have to discuss which is the actual language on which we can express our query q over the nested graph model. The chapter that follows will conclude our theoretical framework by defining such query language, while the second last chapter will focus more on algorithmic and implementation issues concerning an algorithm for a specific instance of the nesting operator, ν.

6 GSQL: a Generalized Semistructured Query Language

Contents

> *Language is a process of free creation; its laws and principles are fixed, but the manner in which the principles of generation are used is free and infinitely varied. Even the interpretation and use of words involves a process of free creation.*
>
> — Noam Chomsky, *Language and Freedom*, (87-8)

The definition of a new data model requires a new query language: even though several distinct algorithms and query languages have been developed distinctly for integrating either graph and semistructured data, the definition of an algebra (and hence, a set of operations) can detect which is the minimal set of the required operations. This thesis also shows that the General Semistructured Query Language proposed at Section 6.1 expresses all the possible queries over different data models over GSM. This query language offers the primary building blocks over which we can **(i)** implement the (semistructured) relational algebra and the nested graph operator (Section 6.3.1 on page 172), and **(ii)** define the semantics for traversal query languages (both semistructured and graph, Section 6.3 on page 172). Section 6.2.2 on page 164 shows that such algebra is able to define two different abstraction operators (α_1, α_2) that can be used in the last example from the previous chapter for extracting a schema from semistructured data sources. The same section is also going to show that these abstraction operators are a specialisation of the nesting operator which, as introduced in Chapter 2, is also able to express the whole class of the $\otimes_\theta$-products and the semistructured grouping γ.

The previous chapter showed that by embedding expressions within the data structure we might achieve structural aggregation as required by the stream data model; this idea is going to be refined in Section 6.3.3, where we also use GSQL to achieve the final goal. The proposed General Semistructured Query Language may also use the former chapter's `script` expressions for providing predicates and functions ($\mathcal{L}_{MM} \equiv \text{GSQL}_{\texttt{script}}$), thus allowing to extend the already-existing data structures with data manipulations.

Last, this chapter is going to show that the combination of GSM and GSQL extends the graph grammar approach as currently implemented in current graph query languages (match + transformation) by also allowing structural aggregation. Such general operator is going to be implemented as one single derived operator in Section 6.3.4. Moreover, we're also going to show that a refinement of this operator allows the definition of the Ϙ operator for data transformation over schema alignments. Moreover, this chapter will also show that traditional traversal languages may be also expressed via GSQL operators such that GSM are closed over the GSQL expressions.

6.1 General Semistructured Query Language (GSQL)

As observed within the dynamic graph context [DEGI10], the four main unary graph operations that a database must support are the insertion, deletion, and the update of the basic components of a data structure, alongside with its navigation (filtering). Please note that, within (nested) graphs, the vertices' and edges' operations must be implemented differently: while the removal of an edge has no consistency problems, the removal of a vertex must imply the removal of all the incoming and outgoing edges and, in this sense, it can be considered as the removal of several elements at once. This observation leads to the fact that we can't unify one same operator for different data constructors. Therefore, the only way to generalise such operations is to generalise and de-specialise the data model, as it has been already done for GSM. Moreover, the graph join contribution in Chapter 4 also showed that binary graph operations can be all implemented with the same subset of relational operations plus some refinements required to preserve the data model's constraints (see Section 6.3.1 on page 172). This intuition can be supported by the GSM model, where both vertices and edges are represented as objects: for example, instead of defining an explicit operation allowing the creation of new edges, we can define an operation creating new objects within the GSM that is also going to be used for the edge creation task. By doing so, the object creation operation may be also used in further several data representations, from relational data, to semistructured and nested graphs represented in GSM. Given that also different models and operations may come after this thesis, we want to provide the most basic set of operations that may be adopted as a common basis by either current or subsequent query languages over GSM.

Object Creation▷

Before providing a definition for nested graphs, we're going to describe the basic operations to be used within the nested semistructured model. We're now going to discuss and motivate the object creation operator: this operation is generally required by our data model, because such model can only express ϕ containment over objects that already exist. Therefore, if we want either to extend or aggregate some objects, we must create first an object representing either its extension or aggregation. The object creation operator can be defined as follows:

▸ **Definition 39** (Object Creation). *Given an GSM $n = (o, O, \ell, \xi, \phi)$, the* ***object creation*** *of $\omega \notin O$ associated to a set of labels L and expressions E with a containment function ϕ_ω for a list of elements already contained in O ($\mathrm{cod}(\phi_\omega) \subseteq \wp(O)$) is defined as follows:*

$$\texttt{create}^{\omega}_{L,E,\phi_\omega}(n) = (o_c, O \cup \{\omega\}, \ell \oplus [[\omega, L]], \xi \oplus [[\omega, E]], \phi \oplus [[\omega, \phi_\omega]])$$

◂

In particular, both ontology alignments and data cleaning processes [SPR17] use the creation of new relationships for storing the similarity between the matched values; such operation is called **link discovery** or **match** (see Section 6.3.1 on page 172).

Object removal▷ *is a specific case of filtering*

The removal of an element from an object (or a collection) is not a primitive operation, because it can be expressed through a filtering predicate expressing the concept "not appearing in". In particular, within the relational model we can express a set difference $A\backslash B$ (stating the removal from A of all the objects contained by B) through a selection predicate as follows:

$$A\backslash B = \{\, x \in A \mid x \notin B \,\} = \sigma_{x \mapsto x \notin B}(A) \tag{6.1}$$

Moreover, the database filtering[1] is the most relevant operation for several traversal and visiting tasks [TPAV17, HG16, MSV17, FPG15]. Nevertheless, the filtering operation can be further on generalized by the definition of a transformation function: given that all the GSM objects can contain other objects associated to properties within ϕ, it implies that the filtering operation over the objects has to be considered as a special case of the "update" or map operation, where the non-desired objects may also be removed. Let us define the map operator first, which will update any object o_c contained in the object repository O by creating a new one with a different id (o_{c+1}) when at least one of his two functions are not involve into a transformation:

◁ *Filtering is a specific case of mapping*

◁*Map operator.*

▸ **Definition 40** (Map). *Given an GSM $n = (o, O, \ell, \xi, \phi)$, the **map** operator $\mathtt{map}_{f_L, f_E, f_C}$ associates to each object o represented in $\varphi^*(o)$, o included, a new one having labels $f_L(o)$, expressions $f_E(o)$ and containments $f_C(o)$. Moreover, it associates a new id to all the transformed objects δO such that $\delta O = \{\, o \in O \mid f_L(o) \neq \ell(o) \vee f_E(o) \neq \xi(o) \vee f_C(o) \neq \phi(o) \,\}$:*

$$\begin{aligned}
\mathtt{map}_{f_L, f_E, f_C}(n) = (o\ _{c+1},\\
&O \cup [o_{c+1} | o_c \in \delta O],\\
&\ell \oplus \bigoplus_{o_c \in \delta O} [[o_{c+1},\ f_L(o_c)]],\\
&\xi \oplus \bigoplus_{o_c \in \delta O} [[o_{c+1},\ f_E(o_c)]],\\
&\phi \oplus \bigoplus_{o_c \in \delta O} [[o_{c+1}, \bigoplus_{e \in \mathrm{dom}(f_C(o_c))} [[e,\ [o'_{c+1} | o'_c \in f_C(o_c, e) \cap \delta O] \cup (f_C(o_c, e) \backslash \delta O)]]]]\\
&)
\end{aligned}$$

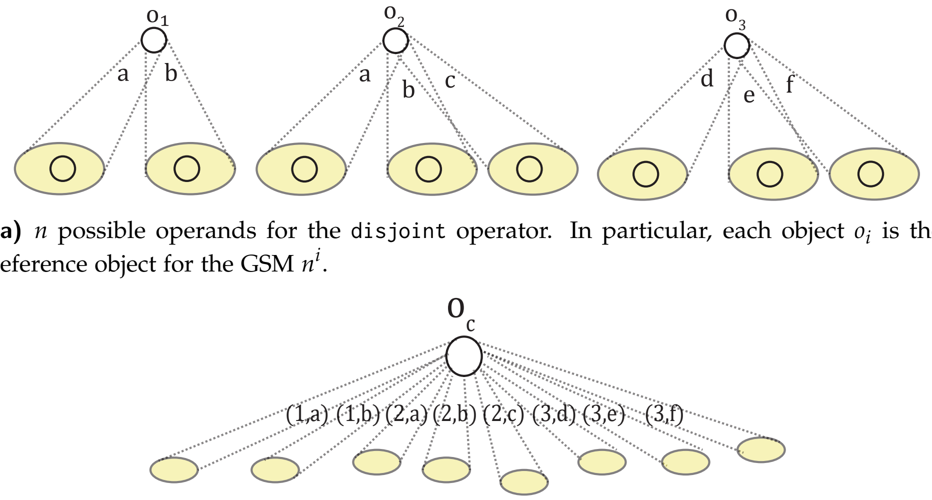
Please note that the double-squared "graph of a function" notation introduced ad page 136 is used to describe the functions' updates. Furthermore, if f_L is defined using `script` *programs ($f_L \equiv [[\mathtt{expr}]]_\Gamma$), the writing $f_L(o_c)$ has to be intended as the evaluation of the associated expression* `expr` *where* `o` *is associated to o_c and* `g` *to o_c ($f_L(o_c) \overset{\text{def}}{=\!=} [[\mathtt{expr}]]_{\Gamma \cup \{(\mathtt{o}, o_c), (\mathtt{g}, o)\}}$). Similar considerations can be also carried out for f_E and f_C. Last, the user must be aware that this operation may transform a nesting-loop free GSM into a general GSM due to the arbitrary way of performing nestings.* ◂

We can now express a filtering operator using `map`: given a GSM $n = (o, O, \ell, \xi, \phi)$, the selection of the objects upon a predicate P within an object containment is expressed by the `map` operator:

$$\mathtt{filter}_P(n) \overset{\text{def}}{=\!=} \mathtt{map}_{[[\mathtt{o.}\mathbf{\mathtt{ell}}]]_\Gamma, [[\mathtt{o.}\mathbf{\mathtt{xi}}]]_\Gamma, [[\mathtt{map((o.phi):\ x\ ->\ [x[0],\ select(x[1]\ :P)])}]]_\Gamma}(n) \tag{6.2}$$

where each expression is going to be evaluated with a context Γ where o is associated to the currently evaluated object and g is n.

[1] Within the context of relational algebra, such operation is referred as *selection* (σ). The term filtering is here used instead of select, because the latter one has a completely different meaning in other query languages, such as SQL, where its remarks a projection operation.



(a) n possible operands for the `disjoint` operator. In particular, each object o_i is the reference object for the GSM n^i.

(b) Result of `disjoint`(n^1, n^2, n^3).

Figure 6.1 `disjoint` operator. As you can see from the picture, each collection from the operand o_i is treated as a separate element of o and, consequently, mapped into the final result o_c.

(S)electing the object over which perform the operations. ▷

These two operators will permit the implementation of most of the unary operators. In order to increase the operations that our language is able to express, we must also include the n-ary operators, taking more than just one data input. Before defining such operator, we must be able to select one specific object within the GSM, over which we're going to perform the n-ary operations in a subsequent step. Most graph and relational n-ary operations are applied within the same database, where we often have to select the operands over which the queries have to be carried out: (e.g. `FROM` in SQL, `GRAPH` in SPARQL). The selection may be implemented by changing (e.g., "electing") the reference object for the current GSM. Therefore, this preliminary election operator for n-ary operations is defined as follows:

▸ **Definition 41** (Elect). *Given an GSM $n = (o, O, \ell, \xi, \phi)$, the* `elect` *operation chooses an object $o' \in O$ to be used as a new object reference for GSM.*

$$\texttt{elect}_{o'}(n) = (o', O, \ell, \xi, \phi)$$

◂

n-ary disjoint union operator ▷

After providing the possibility of electing the GSM's references, we can now define the class of all the non-recursive n-ary operations over GSMs . By taking take n GSMs as inputs, and then mapping their reference objects into one single object, ω, such object is going to have the labels' (and the expression') set as the union of the input reference objects' labels (and expressions') set; the ϕ containment expresses the disjoint concatenation of the incoming objects concatenation, such that $\phi(\omega, [i, l]) := \phi(o_c^i, l)$ for each input reference object o_c^i with $1 \leq i \leq n$. As a last step, the result of the disjoint union can be mapped as we please to implement the desired n-ary operator. Before doing so in the following section, we provide a definition of such operation:

▸ **Definition 42** (n-ary Disjoint Union). *Given n GSMs, (for each $1 \leq i \leq n$, $n_i = (o_c^i, O^i, \ell^i, \xi^i, \phi^i)$) their **$n$-ary disjoint union*** `disjoint`$(n_1, \dots, n_n)$ *is defined as the new object having as reference a new object $\omega \notin \bigcup_i O_i$. If the ω is omitted, then it is generated as $(\max(\overline{o}^1, \dots, \overline{o}^n) + 1)_{max(c^1, \dots, c^n)+1}$, where $\overline{o}^i_{c^i} = \max O^i$ for each $1 \leq i \leq n$. The labels (and expressions) associated to ω are the union* *of the labels (and expressions) associated to all the reference objects, while the collections are also merged by extending the ϕ^i functions.The resulting GSM is defined as follows:*

$$\begin{aligned}\texttt{disjoint}^{\omega}(n^1,\dots,n^n) = (\omega,\; & \bigcup_{1\leq i\leq n} O^i \cup \{\omega\},\\ & \bigoplus_{1\leq i\leq n} \ell^i \oplus [[\omega,\bigcup_{1\leq i\leq n} \ell^i(o_c^i)]]\\ & \bigoplus_{1\leq i\leq n} \xi^i \oplus [[\omega \bigcup_{1\leq i\leq n} \xi^i(o_c^i)]]\\ & \bigoplus_{1\leq i\leq n} \phi^i \oplus [[\omega, \bigoplus_{1\leq i\leq n} \bigoplus_{p\in \mathrm{dom}(\phi(o_c^i))} [[[i,p],\phi(o_c^i,p)]]]])\end{aligned}$$

Later on, we're going to denote $\ell_{n^1,\dots,n^n}$, $\xi_{n^1,\dots,n^n}$ and $\phi_{n_1,\dots,n^n}$ respectively the ℓ, ξ and ϕ functions associated to the GSM resulting from $\texttt{disjoint}^{\omega}(n^1,\dots,n^n)$. ◂

▸ **Example 28.** *Figure 6.1 on the facing page provides an example of how the disjoint union behaves on three distinct data inputs: as we can see, each collection $\phi(o_i,p)$ belonging to the i-th operand is mapped into the collection $\phi(\omega,[i,p])$ of the expected result. This operator allows to keep distinct collections coming from different operands into one single object, and preserves both the operator and the expression from which it comes from.*

◁ *Expressing iterations.*

We have to define an iterative operator similarly to the one defined in [CLNP06, JPT+16]: most data mining algorithms can be expressed as fold operators jointly with the body of the iteration expressed through a GSQL expression, as already introduced in Section 3.1.1.1 on page 58. Given that our query language shall not be able to diverge, we will restrict such operator to a finite recursion. In particular, we're going to use the following high order `fold` operator that can be applied to any kind of collection.

▸ **Definition 43** (High Order Fold Operator)**.** *Given any finite collection S of elements, the* `fold` *operator takes as inputs a collection S of elements of type Σ over which the iteration is performed, an accumulator "α: A" providing the initial value, and a binary function $f:\Sigma\to A\to A$. Starting from the min of S $\min(S)$ (e.g., the first element of a collection),* `fold` *updates α via f to $f(\min(S),\alpha)$, and then iterates the process until all the elements of S are visited. The final value of α is then returned. This function can be recursively written as follows:*

$$\texttt{fold}_{S,f}(\alpha)=\begin{cases}\alpha & S=\varnothing\\ \mathrm{fold}_{S\setminus \min(S),f}(f(\min(S),\alpha)) & oth.\end{cases}$$

Please note that if α is a GSM, f can be even an arbitrary GSQL expression, and hence it can be used for GSQL expressions. Also note that S can be also a nested graph and, in this case, the iteration is performed over the collection of objects of both vertex and edge set, thus providing another nested graph binary operator. ◂

◁ *Views*

Last, GSQL may also support the creation of view by associating a variable to a GSQL expression. This concept is widely supported on both relational [CLNP06, ACPT99, ACPT09] and graph algebras [GRS+16], in order to be later used as other data inputs. We formally define a GSQL expression as follows:

▸ **Definition 44** (GSQL expression). *Given the set $\mathcal{GSM}$ containing all the possible GSM n and a set* Var *of all the possible variables V_i, a GSQL expression* <GSQL> *is defined as follows:*

$$
\begin{aligned}
\texttt{<GSQL>} := {} & n \in \mathcal{GSM} \mid V \in \mathrm{Var} \\
& \mid \texttt{create}^{\omega}_{L,E,\phi_\omega}(\texttt{<GSQL>}) \mid \texttt{map}_{f_L,f_E,f_C}(\texttt{<GSQL>}) \mid \texttt{elect}_o(\texttt{<GSQL>}) \\
& \mid \texttt{disjoint}^{\omega}(\texttt{<GSQL>}(,\texttt{<GSQL>})^*) \mid \texttt{fold}_{S,f}(\texttt{<GSQL>})
\end{aligned}
$$

The semantics associated to the evaluation of a GSQL expression was provided[2] *by the previous operators' definitions (Definitions 39-43). The variable's semantics is defined by the evaluation of the associated GSQL expression, if any.* ◂

6.2 Derived GSQL operators over GSM

The previous section constructively provided the basic building blocks for expressing data operations. The present section will show that those can be used to express all the possible operations on top of GSMs. In particular, we will redefine all the source models' usual operators and provide traversal semantics. We first describe set operators (Subsection 6.2.1) and then we extend them to relational and semistructured operators (Subsection 6.2.2 on page 164). These operators are going to introduce the nesting operator, which is going to express abstraction, grouping and $\otimes_\theta$-products (as well as joins).

6.2.1 (Attribute labelled) Set operations

In this section we're going to define set operations using the disjoint union operator combined to a `map`, given that `map` can represent most of the unary operators. Consequently, for all the binary (or n-ary) set operations, we must use `disjoint` first and then transform the elements. Each binary (or n-ary) is going to be carried out over each operand's reference object. The set operations are going to be carried out over the collections' referenced by the same expression p via $\phi(o_i, p)$ for each $1 \leq i \leq n$.

Please note that in the following definitions we're going to omit the `script` syntax in favour of a more compact mathematical notation. Nevertheless, we're going to provide the script definition of the incoming ψ functions in Appendix C on page 235.

We now introduce the first GSQL set operation, that is the union. We must remark that this operator is anyway different from the traditional union operator defined for sets: while the present operator considers that two element are the same if and only if they are indeed the same object, the set operator considers two elements to be equivalent just if they have the same value. This consideration implies that some relational algebra operators can be defined through the definition of an equivalence predicate between the objects, and thus requiring the definition of the "group by" operator, that we are going to provide in the next subsection.

▸ **Definition 45** (Union). *Given n GSMs $n^i = (o^i_c, O, \ell^i, \xi^i, \phi^i)$ for each $1 \leq i \leq n$, their* ***union*** *$\bigcup_{1\leq i\leq n} n^i$ maps the union of their object into a new reference object ω, where only the resulting reference object is transformed as follows:*

$$\psi_\cup = o \mapsto p \mapsto \bigcup_{1\leq i\leq n} \phi_{n^1\dots n^n}(\omega, [i,p])$$

[2] See also http://rebrand.ly/unibo9e6c3 for its implementation.

Therefore, the result of the union will return a reference object containing either the union of the containments associated to the same attribute p in both operands, or their preservation. The operator can be defined as follows:

$$\bigcup_{1\leq i\leq n}^{\omega} n^i = \texttt{map}_{\ell_{n^1\dots n^n},\xi_{n^1\dots n^n},\phi_{n^1\dots n^n}\oplus\psi_\cup}(\texttt{disjoint}^\omega(n^1,\dots,n^n))$$

◂

We can define the intersection and difference operators similarly to the union as follows:

▸ **Definition 46** (Intersection). *Given n GSMs $n^i = (o_c^i, O^i, \ell^i, \xi^i, \phi^i)$ for each $1 \leq i \leq n$, their **intersection** $\bigcap_{1\leq i\leq n} n^i$ maps the intersection of their object into a new reference object ω, where only the reference object is transformed in the ϕ function as follows:*

$$\psi_\cap = o \mapsto p \mapsto \bigcap_{1\leq i\leq n} \phi_{n^1\dots n^n}(\omega,[i,p])$$

Therefore, the result of the intersection will return a reference object containing the intersection of the containments associated to the same attribute p in both operands. The operator can be defined as follows:

$$\bigcap_{1\leq i\leq n}^{\omega} n^i = \texttt{map}_{\ell_{n^1\dots n^n},\xi_{n^1\dots n^n},\phi_{n^1\dots n^n}\oplus\psi_\cap}(\texttt{disjoint}^\omega(n^1,\dots,n^n))$$

◂

▸ **Definition 47** (Difference). *Given two GSMs $n = (o_c, O, \ell, \xi, \phi)$ and $n' = (o'_c, O', \ell', \xi', \phi')$, their **difference** $n\backslash^\omega n'$ maps the union of their object into a new reference object ω, where only the reference object is transformed in the ϕ function as follows:*

$$\psi_\backslash = o \mapsto p \mapsto \phi_{n,n'}(\omega,[1,p])\backslash\phi_{n,n'}(\omega,[2,p])$$

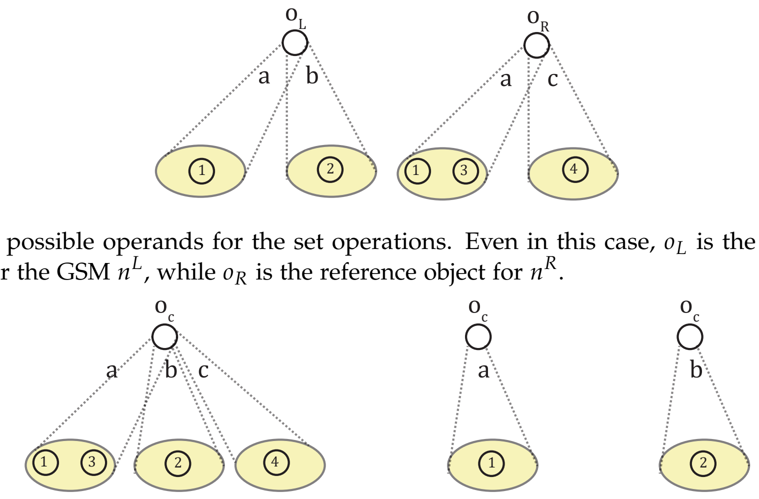
Therefore, the result of the intersection will return a reference object containing the intersection of the containments associated to the same attribute p in both operands, and the remaining elements from the sole left operand:

$$n\backslash^\omega n' = \texttt{map}_{\ell_{nn'},\xi_{nn'},\phi_{nn'}\oplus\psi_\backslash}(\texttt{disjoint}^\omega(n,n'))$$

◂

▸ **Example 29.** *This example provides some graphical representations of the former multi-set operations. Let us take a look at Figure 6.2 on the next page: set operations are performed over the containments both at the collection level (the set operation is performed over all the collections identified by the same attribute) and at the object level. While the union operation (b) merges the collections with the same attribute (containments) coming from both operands, the intersection (c) keeps only the ones appearing in both operands and, while doing so, it preserves only the object shared between the collections. Last, the difference (d) keeps only those collections (containments) appearing on the left operand (n^L) and, among those collections, only the element not appearing in the attribute-correspondent collection in n^R are preserved.*

Given that ϕ associates to each object o the attributes p alongside their list of values $\phi(o,p)$, we can slightly change the previous operations in order to define the combine operation required for the relational join operation, as already outlined in Definition 23 on page 96. As a consequence, even the combine operator may be expressed as the following derived operator.



(a) Two possible operands for the set operations. Even in this case, o_L is the reference object for the GSM n^L, while o_R is the reference object for n^R.

(b) $n^L \cup n^R := \bigcup\{n^L, n^R\}$ **(c)** $n^L \cap n^R := \bigcap\{n^L, n^R\}$ **(d)** $n^L \backslash n^R$

■ **Figure 6.2** Representing different possible outcomes for the set operators. While the `disjoint` operator treated each operand's containment as an element of a set represented by the object itself, these operations treat each containment as a collection. Moreover, the set operations are carried out over the set sharing the same labels.

▸ **Definition 48** (Object Concatenation)**.** *Given two GSMs n and n', their* ***concatenation*** *$n \oplus n'$ maps them into a new reference object ω, such that the attribute-values coming from the first operand are rewritten in favour of the ones coming from the second operand when sharing the same attribute. Therefore, the operator is defined as follows:*

$$n \oplus^{\omega} n' = n' \cup^{\omega} (n \backslash n')$$

◂

The former operation shows that the MOF objects and sets are uniformly represented within this data model through GSM reference objects: this is evident from the definition of the object concatenation operator, which was defined through the composition of over set operators.

6.2.2 Relational and semistructured operations

Attribute ▹ labelled Set operators over relations' GSM representations.

The previous subsection discussed the definition of set (and object) operators on top of GSQL. For this reason, we must make sure that the former definitions are also compliant with GSM translations of relations as defined in Algorithm II.4 on page 137: given that the aforementioned translation maps each possible relation into one object r_o associating all the tuples t_i via a single "Entity"attribute ($t_i \in \phi(r_o, \text{"Entity"})$), by definition of the former operators we have that the desired set operation will be actually performed over the set of objects defined over "Entity". Therefore, the former section provides a straightforward definition for set operations that are compliant with objects produced by a translation from the relational model, and hence they do also provide set operations for the relational model. Moreover, the union actually defines an outer union [LN07], where the resulting relation is the union of two relations' schemas as a resulting schema.

Similar considerations can be also performed for the relational filtering (also known as selection, σ) operator: if we restrict the filtering property just to the elements contained by the object reference in "Entity", we can express the σ_P operator from $\texttt{filter}_P$ presented in Equation 6.2 on page 159 as follows:

$$\texttt{filter}_{\texttt{t -> \{not(t in (g.phi["Entity"])) || } P\texttt{(t)\}}}(n)$$

Please note that such restriction may be completely ignored for semistructured models, where the filter operator may be directly introduced.

At this point, we want to show that the map operator can express other algebraic operators that have already been defined in current literature, such as embedding (ε_{EF}) and the projection (π_{PF}) operators presented in [MM06]. Both operations act as a specific instance of map operators: while EF is defined as a function extending the object's collection with new identifiers or creating new associated collections, PF either reduces the number of collections or reduces their content. Consequently, both operators can be defined as follows:

▸ **Definition 49** (Embedding). *Given a GSM n, its **embedding** is defined as a specific map function EF such that $\forall o \in O. \forall p \in \mathcal{L}. EF(o,p) \supseteq \phi(o,p)$:*

$$\varepsilon_{EF}(n) = \texttt{map}_{\ell_{nn'}, \xi_{nn'}, \phi_{nn'} \oplus EF}(n)$$

◂

▸ **Definition 50** (Projection). *Given a GSM n, its **projection** is defined as a specific map function PF such that $\forall o \in O. \forall p \in \mathcal{L}. EF(o,p) \subseteq \phi(o,p)$:*

$$\pi_{PF}(n) = \texttt{map}_{\ell_{nn'}, \xi_{nn'}, EF}(n)$$

◂

In order to express π as the one defined within the relational model, we can simply express PF as the relevant L attributes for ϕ that must be returned, instead of specifying the whole PF definition ($\forall o \in O. \forall l \in L. l \simeq \phi(o,l)$). If we want to explicitly create new objects resulting from an expression evaluation f returning the list of values to be associated via ξ, we can define the $Calc$ operator [CLNP06] as follows:

▸ **Definition 51** (Calc). *Given a GSM n, the Calc operator extends each object x appearing in $\phi(\mathrm{g}, K_1)$ with a newly created object $(o+1)_c$ contained in $\phi(o, K_2)$; $(o+1)_c$ will have a label set A and value $f(x)$:*

$$Calc^{K_1,K_2}_{f\ \texttt{as}\ \texttt{A}}(n) = \texttt{fold}_{[\![\texttt{o.}\mathbf{\texttt{phi}}\texttt{[K]}]\!]_{\{(\mathrm{g},\mathrm{g}),(\mathrm{K},K_1)\}}, f_K}(n)$$

where $f : O \mapsto \wp(MM)$ and f_K is the accumulation function which is defined as follows:

$$f_K = x \mapsto \alpha \mapsto \textbf{\textit{let}}\ o_c \overset{\text{def}}{=} \max O\ \textbf{\textit{in}}\ \texttt{map}_{[\![\texttt{o.}\mathbf{\texttt{ell}}]\!]_\varnothing, [\![\texttt{o.}\mathbf{\texttt{xi}}]\!]_\varnothing, [\![\texttt{e}]\!]_{(\text{oc},(o+1)_c),(\text{K},K_2),(\text{x},x)}}\left(\texttt{create}^{(o+1)_c}_{A, f(x), \varnothing}(\alpha)\right)$$

and e *is the expression performing the K_2 extension for each x as follows:*

```
map(o.phi : z ↦ {if (o.id == x.id && z[0] == K) then {z[0], z[1] @ {oc}} else
                                              z})
```

◂

If n is a relation obtained using the canonical transformation (Algorithm II.4 on page 137) where each relation contains its tuples in $\phi(\mathrm{g}, \text{"Entity"})$ and each tuple in is represented by an object o containing its attribute in $\phi(\mathrm{g},$“Attribute”$)$, we have that $K_1 \overset{\text{def}}{=}$ “Entity” and that $K_2 \overset{\text{def}}{=}$ “Attribute”.

Expressing the class of nesting operations. ▷ In Section 2.2.2 on page 40 we observed that a nesting operator (Equation 2.4 on page 40) can be used to generalize both joins and grouping operations. As promised, this operator will generalize the already-existing join and grouping operations by expressing the broader class of clustering algorithms which are both overlapping and partial [TSK05], thus allowing the social network clustering scenario. As a consequence, the (derived) nesting operator is going to provide several different data operators, as well as the class of the $\otimes_\theta$-products. As a first intuition, this operator should generalize the class of the group-by operations: therefore, such operation should use an object classifier GF mapping each object into a (possibly empty) subset of clusters in $\mathcal{C}$ ($GF{:}\,O \mapsto \wp(\mathcal{C})$). As showed in the part-of aggregation example (see Example 26 on page 141), the clusters can be summarized into one single object: therefore, the desired operation shall generalize most of the algebraic grouping operations.

This operation requires an expression $\oplus_f$ (Equation 2.3 on page 40) providing a aggregation over either similar or equivalent elements, and a way to generate collection of collections from a initial collection: this last step must use the aforementioned clustering operation GF. The definition of such operator allows to group all the elements belonging to the same cluster and leaves out all the non-represented outliers, that may be included or not in the final result.

As previously discussed for both the ε and $Calc$ operators, this data model does not allow to refer to elements that still do not exist. For this reason each element expressing the result of an aggregation must be created before effectively aggregating the desired components. Moreover, in order to create multiple elements, we must finally iterate over the all possible clusters minable within each object's collection, and then detect which are the group to be created. The set over which the iteration is going to be performed is defined as follows:

$$S_{GF,n} \overset{\text{def}}{=} \left\{ (o,p,k) \middle| o \in \varphi(n), p \in \mathrm{dom}(\phi(o)), \phi(o,p) \neq \varnothing, k \in \bigcup_{o'\phi(o,p)} GF(o,p,o') \right\}$$

Each triplet (o,p,k) contained in this set associates to each non-empty collection $\phi(o,p)$ a labelled cluster $k \in \mathcal{C}$ containing at least one element of $\phi(o,p)$. Therefore, all the elements matched by k in $\phi(o,p)$ are going to be replaced by one single object. This new object has to be created by the $\oplus_f$ function (also mentioned in the third chapter) as follows:

$$\oplus_f(\alpha,k,\omega,\{\, o' \in \phi(o,p) \mid k \in GF(o,p,o') \,\})$$

where α is a previous step of n where another object pointed by $S_{GF,n}$ was generated, and ω is the object id associated to the object generated by the expression associated to $\oplus_f$. In particular we can arbitrarily choose to set ω using an id generation function $gen(o_{\tilde{c}},k,o,p)$, where $o_{\tilde{c}} = \max n.O$. The newly created object by $\oplus_f$ can be concatenated to the object generator via $S_{GF,n}$ and a fold iterator:

$$ce \overset{\text{def}}{=\!=} \mathtt{fold}_{S_{GF,n},(o,p,k)\mapsto\alpha\mapsto\textbf{let }o_{\tilde{c}}=\max O\textbf{ in }\oplus_f(\alpha,k,gen(o_{\tilde{c}},k,o,p),\{\,o'\in\phi(o,p)\,\mid\,k\in GF(o,p,o')\,\})}(n) \tag{6.3}$$

where n provides the initialization of the accumulator for the fold operation. After creating the objects associated to the mined clusters, we can now replace the objects of $\phi(o,p)$ belonging to a cluster k by using a map, which retrieves the clustered objects via gen.

All those intermediate computational steps may be chained together into the following definition of a nesting operator:

▸ **Definition 52** (Nesting). *Given a GSM* $\mathfrak{n} = (\mathfrak{g}, O, \ell, \xi, \phi)$*, the* ***nesting*** *operator* $\nu^{\textbf{\textit{keep}}}_{GF,gen,\oplus_f}$ *aggregates the elements within each non empty collection* $\phi(o,p)$ *(where* $o \in O$ *and* $p \in \mathrm{dom}(\phi(o))$*) by replacing with one single object all the elements belonging to the same class* $k \in \mathrm{cod}(GF)$*. Moreover, all the elements that belong to no class are not aggregated. The outliers may be returned (****keep****=*`tt`*) or not (****keep****=*`ff`*), dependingly on the desired final representation. The operator is defined as follows:*

$$\nu^{\textbf{\textit{keep}}}_{GF,gen,\oplus_f}(n) = \texttt{map}_{\ell,\xi,o\mapsto p\mapsto[o'\in\phi(o,p)|GF(o,p,o')=\emptyset\wedge\textbf{\textit{keep}}]\,\cup\,\bigcup[gen(o_{\tilde{c}},k,o,p)|k\in\bigcup_{(o,p,o')\in\mathrm{dom}(GF)}GF(o,p,o')]}(ce)$$

where "ce" was defined in Equation 6.3 on the facing page. This operation uses the gen function to associate the associations and classes generated by GF to the objects that are generated in the "ce" phase. ◂

Given that this operation can replace any object within any collection, GF can be constrained within the relational model by ensuring that GF must return an empty set for any (o, p, k) where o does not appear as an entity within n. This approach is similar to what it has been previously stated for `filter`. The former definition (and restrictions) allows to instantiate the other derived operators, for both relational and semistructured models.

Before introducing some of the possible derivations for such operator, we introduce the last remaining operator, which is the opposite operation of nesting: the unnesting operation. In this case, we must select which element $o' \in \phi(o, p)$, within a given object o and associated to an attribute p, has to be replaced by its expansion in $\phi(o', p')$, where p' is an attribute appearing in the given set. Given that now the unnesting choice is binary, we're going to select which elements are going to be expanded, dependingly to the attribute p where o' is contained and on o' itself.

▸ **Definition 53** (Unnesting). *Given a GSM* n*, a set of attributes* $a \in A$ *– over which replace and expand via the objects* o' $(\phi(o', a))$ *that are contained in* $o' \in \phi(o, p)$ *–, and a binary predicate* P *through which select the* o' *appearing in* p $(P(p, o'))$*, the* ***unnesting*** *operator is defined as follows:*

$$\mu_{A,P}(n) = \texttt{map}_{\ell,\xi,o\mapsto p\mapsto[o'\in\phi(o,p)|\neg P(p,o')]\,\cup\,\bigcup[\phi(o',p')|p'\in A,o'\in\phi(o,p),P(p,o')]}(n)$$

◂

As we will see in the following four operators, relational joins, grouping and abstraction operators may be all derived from the nesting operator.

Data-Preserving Aggregation (α_2)

We can generalize the aggregation operator by associating all the GF-similar elements to one single element, containing all the references. In particular, we use the ɤ[3] *gen* function for generating new ids:

$$\text{ɤ}(o_{\tilde{c}}, k, o, p) = \{\, (o + dtl([\texttt{bin}(k), o, \texttt{bin}(p)]))_{\tilde{c}+1} \,\}$$

[3] This symbol is called *ram's horns* in linguistics.

where `bin` is the function associating to each element its byte representation expressed as `bigint` compatible with the *id* definition. By using the dovetailing function over the binary representation of the triplet, we ensure that different *id*-s are going to be associated to their correspondent generated objects. We let the user decide on how to represent the resulting set of labels, the set of expressions, and the containment respectively through f_L, f_E and f_C functions.

$$\alpha_2{}^{\mathbf{keep}}_{GF,f_L,f_E,f_C}(n) = \nu^{\mathbf{keep}}_{GF,\text{ɤ},(\alpha,k,\omega,s)\mapsto\texttt{create}^{\omega}_{f_L(k,s),f_E(k,s),f_C(k,s)}(\alpha)}(n)$$

▸ **Example 30.** *In Example 27 on page 144 we addressed the problem of extracting the schema from a JSON representation of a graph. As we previously outlined, we could choose to use an aggregation where each matched component via a class k is nested within an object o via k, and that k is used as a label for the object that will contain such data. Therefore, the desired result can be achieved via the following application of the α_2 aggregator:*

$$\alpha_2{}^{\texttt{tt}}_{GF_1,(k,s)\mapsto[k],(k,s)\mapsto f(s),(k,s)\mapsto[[k,s]]}(n)$$

At this point we want to aggregate each object by its associated label; if the object is a "label" object, we want to return the value associated to it designign the containing object's label; if the object ha associated to a non relevant label w.r.t. the schema extraction process (e.g. "metadata"), a set containing the concatenation of the GF classes of all the concatenating object is returned; in all the other cases, no nesting is performed. the following clustering operation describes the desired result:

$$GF_1(\_,\_,o) = \begin{cases} \varnothing & \text{“metadata”} \in \ell(o) \lor P(o) \\ \xi(o) & \text{“label”} \in \ell(o) \\ \ell(o) & \ell(o) \neq \varnothing \\ \varnothing & \text{oth.} \end{cases} \tag{6.4}$$

where the underscores remark the ignored arguments. P is a predicate avoiding to aggregate the elements that are represented only once within the hierarchy and appear at the coarsest levels of it; such predicate is defined as follows:

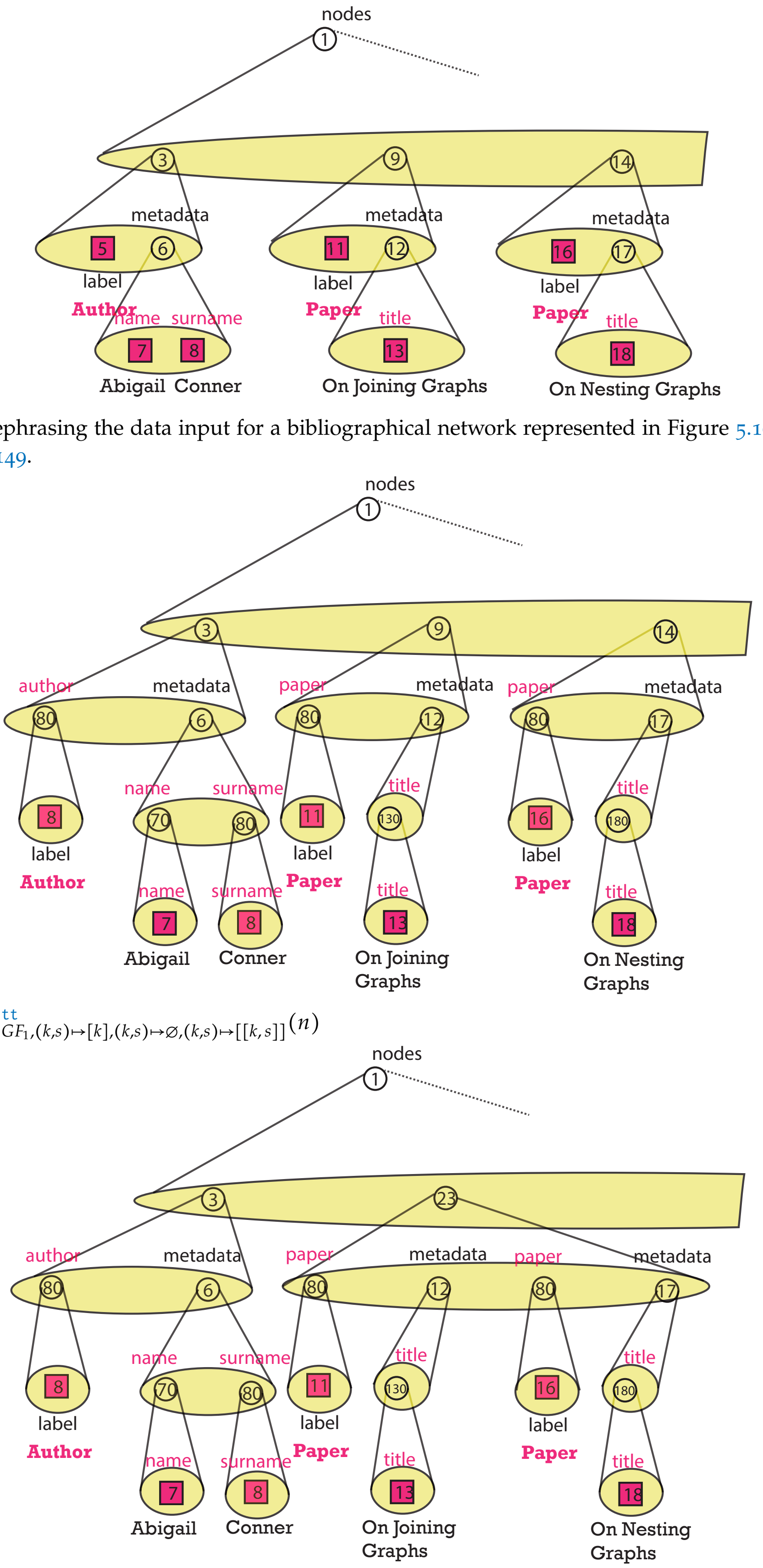
$$P(x) = (\ell(x) = \varnothing \lor (\neg\exists o' \in \varphi^*(o.n).o' \neq x \land \ell(o') = \ell(x))) \land (\forall o' \in \varphi^*(o.n).x \in \varphi^+(o') \Rightarrow P(o'))$$

The result of the application of such aggregation to the nested structure represented in Figure 6.3a is provided in Figure 6.3b: this operation preserves all the original data within each aggregated element but, at the same time, increases the amount of generated data. As a consequence, this solution could potentially increase the time required to visit the data structure that may occur at subsequent steps. Therefore, a different approach preserving the GSM height in spite of the representation of the original information is required.

Grouping (γ)

The grouping operation for semistructured data was originally presented in [MM06], where $\oplus_f$ is simply defined as the n-ary union of all the matched objects, thus allowing to integrate each similar component into one single representation. The desired operation can be described as follows:

$$\gamma^{\mathbf{keep}}_{GF}(n) = \nu^{\mathbf{keep}}_{GF,\text{ɤ},(\alpha,k,\tilde{o}_c,s)\mapsto\texttt{elect}_o\left(\bigcup_{i\in s}^{\tilde{o}_c}\texttt{elect}_i(\alpha)\right)}(n) \tag{6.5}$$



(a) Rephrasing the data input for a bibliographical network represented in Figure 5.10a on page 149.

(b) $\alpha 2^{\mathtt{tt}}_{GF_1,(k,s)\mapsto[k],(k,s)\mapsto\varnothing,(k,s)\mapsto[[k,s]]}(n)$

(c) $\mathtt{map}_{x\mapsto\ell(x)\backslash\text{``}\alpha\text{''},\xi,\phi}(\gamma^{\mathtt{tt}}_{GF_2}(\alpha 2^{\mathtt{tt}}_{GF_1,(k,s)\mapsto[k,\text{``}\alpha\text{''}],(k,s)\mapsto\varnothing,(k,s)\mapsto[[k,s]]}(n)))$

Figure 6.3 Representing different possible results from the application of the general nesting definition. *(cont.)*

▸ **Example 30** (continuing from p. 168). *Figure 6.3c shows an example of grouping. In this case we want to aggregate even the elements that were not previously inserted within a cluster. Therefore, we change the α_2 by marking with "α" the elements matched by GF_1. Last, the following GF function for γ is provided:*

$$GF_2(\_,\_,o) = \begin{cases} \ell(o)\backslash\{\text{``}\alpha\text{''}\} & \text{``}\alpha\text{''} \in \ell(o) \\ \ell(o) & \ell(o) \neq \varnothing \wedge \exists o' \in \varphi^{+}(o).GF_2(\_,\_,o') \neq \varnothing \\ \bigodot_{o' \in \varphi(o)} GF_2(\_,\_,o') & \ell(o) = \varnothing \\ \varnothing & \text{oth.} \end{cases}$$

where $\bigodot$ is the string concatenation function over set of strings. As we can see, this operation does not structurally propagate the aggregation within all the containment levels, but it only aggregates the data at the first nesting level available. In order to propagate the nesting in depth, we must iterate the same operation until all the similar components are structurally aggregated together. Therefore, it is now relevant why the `fold` *construct is relevant for our algebra.*

Abstraction (α_1)

Let us now discuss on how to achieve the α schema extraction operator over our nested data representation. In the previous chapter we mentioned that, within this thesis, we are going to work exclusively on nesting-loop free GSMs: this constraint allows a definition of structural length of a GSM. Given that the structured aggregation must be further propagated towards the leaves after each iteration, we can consider the GSM's height as an upper bound to the number of iterations required to propagate the aggregations as expected. Therefore, we may first perform an α_2 aggregation and, after that, we can recursively group by the former labels. Such operator may be defined as follows:

▸ **Definition 54** (Structural Aggregation). *Given an equivalence relation SF to be tested among the objects within each containment $\phi(o,p)$, the structural aggregation propagates the aggregation result to all the underlying data structures by marking them with a "α" label. This operator is then defined as follows:*

$$\alpha_1{}_{SF}^{\textbf{keep}}(n) = \gamma_{GF_2}^{\textbf{keep}}(\alpha)(\texttt{fold}_{\{\, i\in\mathbb{N} \mid 0<i<h(n)\,\},x\mapsto\alpha\mapsto\gamma^{\texttt{tt}}_{GF_2}(\alpha)}(\alpha_2{}^{\texttt{tt}}_{SF,(k,s)\mapsto[k,\text{``}\alpha\text{''}],(k,s)\mapsto\varnothing,(k,s)\mapsto[[k,s]]}(n)))$$

◂

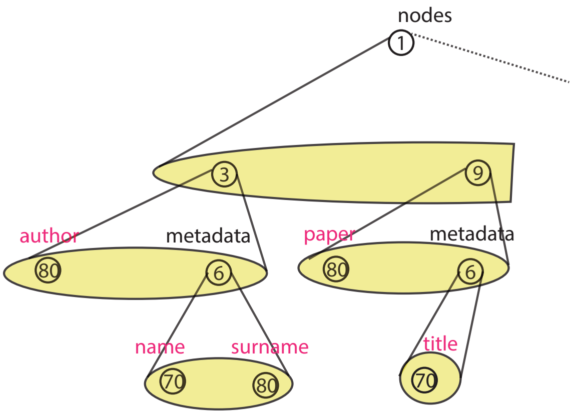
▸ **Example 30** (continuing from p. 170). *Figure 6.3d on the next page provides the desired solution allowing to implement the schema extraction operation outlined in Example 27 on page 144 for the data integration scenario. In order to met the requirements of the former definition, GF_1 presented in Equation 6.4 on page 168 is used for the first aggregation step, while the remaining ones are performed via the following function applied to has to be extended in order to mark the elements that must be structurally aggregated.*

After performing this operation, we can now aggregate the left out elements, thus allowing to implement the relational group over overlapping classes.

▸ **Definition 55** (Multi Group-By). *Given an equivalence relation SF to be tested among the objects within each containment $\phi(o,p)$ and an aggregation function expressed by the three functions f_L, f_E, f_C to be applied over the remaining non-aggregated objects via SF, the **multi group by** over a GSM n is defined as follows:*

$$\Gamma_{SF}^{f_L,f_E,f_C}(n) = \alpha_2{}^{\texttt{tt},f_L,f_E,f_C}_{(o,p,o')\mapsto\text{``}\alpha\text{''}\notin\ell(o')?\text{``}\alpha\text{''}:\varnothing}(\alpha_1{}^{\texttt{tt}}_{SF}(n))$$

◂



(d) $\mathtt{map}_{x\mapsto\ell(x)\backslash``\alpha",\xi,\phi}(\alpha 1^{\mathsf{ff}}_{GF_1}(n))$

Figure 6.3 Representing different possible results from the application of the general nesting definition.

$\otimes\theta$-Product

Given that $\otimes\theta$-products take two GSMs as an input and provide one single GSM, we must preliminarily merge the two operands n and n' via a disjoint union $\mathtt{disjoint}^{\omega_c}(n,n')$. Consequently, the input GSM's reference objects are considered as multiple labelled sets and, therefore, the final join operation shall be performed among all the collections appearing in the left and right operand. In particular, for each collection $\phi(\omega_c,[1,p'])$ and $\phi(\omega_c,[2,p''])$ respectively coming from the first and second operand, we want to return a new collection $\phi(\omega_{c+1},p'p'')$ containing the result of the $\otimes_\theta$ product of the contained objects. As a final result, a `create` predisposing the $p'p''$ collection has to be performed immediately after the disjoint union.

The clustering function GF shall create a distinct cluster for each pair of matching objects x and y, respectively contained in the collections $\phi(\omega_c,p')$ and $\phi(\omega_c,p'')$, and then associates each object to the pairs $\mathtt{bin}(p')\mathtt{bin}(p'')xy$. Such set of elements will be used in the clustering function (Equation 6.6a) and can be defined as follows:

$$\begin{aligned}JS^{\mathbf{c}}_{\omega_c,p',p''}(x) =&\{(\mathtt{bin}(p')\cdot\mathtt{0}\cdot\mathtt{bin}(p'')xy)_{c+1}|\theta(x,y), x\in\phi(\omega_c,[1,p']), y\in\phi(\omega_c,[2,p''])\}\\ &\cup\{(\mathtt{bin}(p')\cdot\mathtt{0}\cdot\mathtt{bin}(p'')yx)_{c+1}|\theta(y,x), x\in\phi(\omega_c,[2,p'']), y\in\phi(\omega_c,[1,p'])\}\end{aligned}$$

On the other hand, each $[1,p]$ and $[2,p'']$ collection contained in $\mathtt{disjoint}^{\omega}(n',n'')$ must be empty (Equation 6.6c) while all the remaining objects' containment should be kept unaltered (Equation 6.6b).The whole definition of GF is defined as follows:

$$GF_\theta(o,p,x)=\begin{cases}JS^{\mathbf{c}}_{\omega_c,p',p''}(x) & o=\omega_{c+1}, p=p'p''.[1,p'],[2,p'']\in\mathrm{dom}(\phi(\omega_c))\text{(6.6a)}\\ \bot & o\neq\omega_{c+1} \qquad\qquad\qquad\qquad\qquad\qquad\text{(6.6b)}\\ \varnothing & \text{oth.} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{(6.6c)}\end{cases}$$

This definition is then involved in two different roles: first, *(i)* GF_θ is used in Equation 6.3 on page 166 to generate the pairs $(o,p'p'',\mathtt{bin}(p')\cdot\mathtt{0}\cdot\mathtt{bin}(p'')xy)$, so that the aggregation function $J_\otimes$ used for $\oplus_f$ is able to generate a new object by $\otimes$-concatenating the two objects u and v matching with predicate θ for containments b_1 and b_2:

$$J_\otimes(\alpha,\mathtt{bin}(p')\cdot\mathtt{0}\cdot\mathtt{bin}(p'')xy,\omega,s)=\mathtt{elect}_\omega\left(\bigotimes_{u\in s}^{\mathtt{bin}(p')\cdot\mathtt{0}\cdot\mathtt{bin}(p'')xy}\mathtt{elect}_u(\alpha)\right)$$

where $\otimes$ is a generic aggregation operation for each element u appearing in the cluster for the elements matching the cluster label $\texttt{bin}(p')\cdot\texttt{0}\cdot\texttt{bin}(p'')xy$ that is going to be used for an object identifier for the previously-aggregated element. Then, the operators re-set ω as a reference object over which perform the remaining operations. For the relational join purposes, we can choose the previously-introduced concatenation operator $\oplus$ as $\otimes$ for combining the matched objects.

Last, ***(ii)*** the JS function within the GF_θ classifier is used in cooperation with γ' to define which are the resulting containments, directly generated by GF_θ as a result of the $\otimes\theta$-product operator, and which are the elements that are not involved by the $\otimes\theta$-product operation ($o_{\tilde{c}} \neq \omega$); while in the first case the result of the $\otimes_\theta$ combination shall be returned for each matched pair of objects (Equation 6.7a), for the other cases where the resulting reference object is not involved and hence the involved map must neither change nor update their containments must be preserved (Equation 6.7b).

$$\gamma'(o_{\tilde{c}}, k, o, p) = \begin{cases} \phi(o,p) & k = \bot \qquad (6.7a) \\ \{k\} & \text{oth.} \qquad (6.7b) \end{cases}$$

Consequently, the binary $\otimes\theta$-product operation can be defined as follows:

$$n \otimes_{JS,\theta} n' = \nu^{\mathsf{ff}}_{GF_\theta,\gamma',J_\otimes}\Big(\texttt{create}^{\omega_{c+1}}_{\ell(\omega_c),\xi(\omega_c),\oplus_{\substack{[1,p'],\\ [2,p'']\in\mathrm{dom}(\phi(\omega_c))}}[[p'p'',\phi(\omega_c,[1,p'])\cup\phi(\omega_c,[2,p''])]]}\Big(\\ \texttt{disjoint}^{\omega_c}(n,n') \\ \Big)\Big)$$

Consequently, by replacing $\otimes$ with $\oplus$ we achieve and by constraining θ to check that all the elements contained by the to-be-merged elements and having the same ℓ label must show the same ξ values, we have the implementation of the join operator, thus the following expression provides the join definition alongside the definition of the restriction for θ:

$$n \bowtie_\theta n' \overset{\text{def}}{=} n \oplus_{\substack{JS^c, \quad (x,y)\mapsto\theta(x,y)\wedge\forall e\in\mathrm{dom}(\phi(x))\cap\mathrm{dom}(\phi(y)). \\ \forall x'\in\phi(x,e).\forall y'\in\phi(y,e). \\ \ell(x')\cap\ell(y')\neq\varnothing\Rightarrow\xi(x')=\xi(y')=\varnothing\vee\xi(x')\cap\xi(y')\neq\varnothing.}} n'$$

6.3 GSQL Use cases

The previous section showed that GSQL is able to express set, relational and semistructured operators. This section shows that such minimal operators' set may be also adopted for **(i)** providing nested graph operators including data mining manipulating ones (Subsection 6.3.1), **(ii)** providing a semantics for graphs and semistructured traversal languages (Subsection 6.3.2 on page 177), **(iii)** performing **is-a** aggregations by combining nested graphs with semistructured hierarchies (Subsection 6.3.3 on page 182), **(iv)** and implementing the definition of a generalized graph grammar operator for nested graphs (Subsection 6.3.4 on page 183). Such definitions may adopt basic GSQL operator or may require derived GSQL operators.

6.3.1 paNGRAm: Nested Graph Relational Algebra

Previously we defined set of operations to be applied over GSM data representation that have no specific domain constraints. One of the data structures that have such kind of constraints are graphs: an edge cannot generally exist without either a source or a target vertex. Therefore, while defining such operations, we must also make sure that the original model constraints' are preserved. This also implies that some operations, previously defined for general GSM, will be implemented for nested graphs with some restrictions (`constraint`). The following paragraphs group the required operations by type.

Unary operators

◁ *Vertex and Edge creation*

Walking on the footsteps of GSQL, we have to define two different creation operations for both vertices and edges because they have two different requirements, while other value creation operations may directly use the `create` operation. Concerning the vertices, we have that vertices are specific object contained in a graph's "Entity" collection by definition. It follows that the creation of an object is not a sufficient definition for their creation. Therefore, we must chain the creation operation immediately with a map operation, allowing to immediately insert the object inside a graph (in this case, the reference object).

▸ **Definition 56** (Vertex Creation). *Given a nested graph $\eta = (\mathfrak{g}_c, O, \ell, \xi, \phi)$, the* ***vertex creation*** *operator creates and promotes an object with a fresh id ω into a vertex by embedding it into a graph $\mathfrak{g}' \in \varphi(\mathfrak{g})$. By knowing the object that will contain ω as a vertex in $\mathfrak{g}'$, we can also apply some restrictions to the contents of f_C that is going to define $\phi(\omega)$ by removing from its codomain all the elements containing $\mathfrak{g}'$ and $\mathfrak{g}'$ itself (as well as ω).*

$$\kappa^{\mathfrak{g}' \leftarrow \omega}_{L,F,f_C}(\eta) = \texttt{map}_{\ell,\xi,\phi \oplus \mathfrak{g}' \mapsto \text{“Entity”} \mapsto \phi(\mathfrak{g}', \text{“Entity”}) \cup [\omega]}(\texttt{create}^{\omega}_{L,F,x \mapsto f_C(x) \backslash [\omega, o' \in O | \mathfrak{g}' \in \varphi^*(o')]}(\eta))$$

◂

We can follow the same approach for creating edges: in this case we can generalize the creation of one single element into a creation of several edges, so that the "link discovery" class of operations can be implemented [NHNR17] via the straightforward satisfaction of a θ predicate among the vertices.

▸ **Definition 57** (Edge Creation (Link Discovery)). *Given a nested graph $\eta = (\mathfrak{g}_c, O, \ell, \xi, \phi)$, the* ***edge creation (link discovery)*** *operator $\dot{\kappa}^{\theta}_{L,F,f_C}$ establishes new edge d_{u_c,v_c} s.t. $\lambda(d_{u_c,v_c}) = (u_c, v_c) \in \phi(\mathfrak{g}, \text{“Entity”})^2$ having L as a labels set and $F(u_c, v_c)$ as a set of expressions. Such edge links two vercies satisfying the given θ predicate. Such operator is defined as follows:*

$$\dot{\kappa}^{\mathfrak{g}' \Leftarrow \theta}_{L,F,f_C}(\eta) = \texttt{map}_{\ell_{\eta\eta'}, \xi_{\eta\eta'}, \phi_{\eta\eta'} \oplus [[\mathfrak{g}', \phi(\mathfrak{g}') \oplus [[\text{“Relationship”}, \phi(\mathfrak{g}_c, \text{“Relationship”}) \cup [d_{u_c,v_c} | (u_c,v_c) \in S_\theta]]]]]}(\texttt{fold}_{S_\theta, f_3}(\eta))$$

where S is the set containing all the pair object-vertices satisfying the θ predicate:

$$S_\theta \overset{\text{def}}{=} \left\{ (u_c, v_c) \in \phi(\mathfrak{g}, \text{“Entity”})^2 \,\middle|\, \theta(u_c, v_c) \right\}$$

and f_3 is the function creating the new edge for each pair in S_θ:

$$f_3 \overset{\text{def}}{=} ((u_c, v_c), \eta') \mapsto \texttt{create}^{d_{u_c,v_c}}_{L, F(u_c,v_c), (x \mapsto f_C(x) \backslash [\omega, o' \in O | \mathfrak{g}' \in \varphi^*(\mathfrak{g})]) \oplus [[\text{“src”}, [u_{c+1}]], [\text{“dst”}, [v_{c+1}]]]}(\eta')$$

d_{u_c,v_c} is the function generating the new edge id from each pair of vertices u_c and v_c defined as $(\max O + dt(u,v))_c$. By knowing the object that will contain ω as an edge in $\mathfrak{g}'$, we can also apply some restrictions to the containts of f_C that is going to define $\phi(\omega)$ by removing from its codomain all the elements containing $\mathfrak{g}'$ and $\mathfrak{g}'$ itself (as well as ω). ◂

Please note that `create` (Definition 39 on page 158) can be always used to create other non-vertex or non-edge objects. Let us now discuss the `map` operator for GSMs for transforming vertices and edges: as we previously discussed, a general embedding function f_C can potentially undermine the nested graph model constraints; therefore, even if `map` (Definition 40 on page 159) will be still used for defining other nested graph operations, we may restrict the `map` operation to the sole label and value update for consistency reasons:

Update ◁

▸ **Definition 58** ((Value and Label) Update)**.** *Given a nested graph $\eta = (\mathfrak{g}_c, O, \ell, \xi, \phi)$, the **(value and label) update** operator*[4] υ_{f_L,f_E} *associates to each object o represented in $\varphi^*(\mathfrak{g})$, $\mathfrak{g}$ included, a new one having labels $f_L(o)$ and expressions $f_E(o)$. Moreover, it associates a new id to all the transformed objects δO such that $\delta O = \{\, o \in O \mid f_L(o) \neq \ell(o) \vee f_E(o) \neq \xi(o) \,\}$. The operator is defined as follows:*

$$\upsilon_{f_L,f_E}(\eta) = \mathtt{map}_{f_L,f_E,\phi}(\eta)$$

◂

This decision also implies that filter operations are not directly possible within this model. As we previously observed, such operations will be generalized through (nested graph) traversal operations as observed in the previous section for NautiLOD, through which edges may also be removed using a GSM traversing semantics.

elect, fold, Calc. ▷

Continuing in the same steps of GSQL, we can `elect` the graph to be used as a reference object. Please note that given that the GSM nesting-loop freeness condition applies to any object of the GSM, the `elect` operation does not undermines the model constraints for the GSM. Therefore, the very same operator in Definition 41 on page 160 may be used. Similar considerations may be also applied to the `fold` (Definition 43 on page 161) and *Calc* (Definition 51 on page 165).

Before defining the unnesting operator, we must observe that we must check whether the result of the unnesting operator returns a correct nested graph. Therefore, we must define this intermediate operator assuring the consistency:

▸ **Definition 59** (Nested Graph Constraint)**.** *The* `constraint` *operation over a nested graph η assures that all the edges appears as nodes within one of the objects of $\varphi^*(\mathfrak{g})$. This operator can be defined via a* `map` *operator as follows:*

$$\mathtt{constraint}(\eta) = \mathtt{map}_{\ell,\xi,\phi\oplus[[\mathfrak{g},[{}^{[\text{“Entity”},\phi(\mathfrak{g},\text{“Entity”})],}_{[\text{“Relationship”},[e\in\phi(\mathfrak{g},\text{“Relationship”})|\exists o,o'\in\varphi(\mathfrak{g}).\lambda(e)\in\phi(o,\text{“Entity”})\times\phi(o',\text{“Entity”})]]}]]]}(\eta)$$

◂

Please note that a combination of `constraint` with a `fold` recursion may also ensure that each graph within the nested graph η satisfies the model's requirements. Moreover, observe the similarity between the former operator and the edge restrictions for graph grammars as in Definition 20 on page 80. At this point, we can define the unnesting operator, which replaces the nested vertices and edges within the reference object with their content:

$$\mu_P^{\mathbf{keep}}(\eta) = \mathtt{constraint}\Bigg($$

$$\mathtt{map}_{\ell,\xi,\phi\oplus\left[\left[\mathfrak{g},\left[{}^{[\text{“Entity”},[o'\in\phi(\mathfrak{g},\text{“Entity”})|\neg P(o')\wedge\mathbf{keep}]\cup\bigcup_{o\in\phi(\mathfrak{g},\text{“Entity”}),P(o)}\phi(o,\text{“Entity”})],}_{[\text{“Relationship”},[o'\in\phi(\mathfrak{g},\text{“Relationship”})|\neg P(o')\wedge\mathbf{keep}]\cup\bigcup_{o\in\phi(\mathfrak{g},\text{“Relationship”}),P(o)}\phi(o,\text{“Relationship”})]}\right]\right]\right]}(\eta)\Bigg)$$

[4] It is represented by the `upsilon` greek letter, Υ, υ.

Finally, the (nested) graph operators may be defined as a generalization of the semistructured nesting approach for arbitrairly aggregating vertices and edges together in similar clusters, that then are going to be recombined:

▸ **Definition 60** (Graph Nesting). *For every nested graph η, given a clustering function CL returning set of pairs $k = (a, b)$, where b is the nested element identifier, either vertex ($a = \mathtt{e}$) or edge ($a = \mathtt{r}$) in which the element is going to be nested to and two (pairs of) transcoding functions $UDF_V = \langle f_E^V, f_C^V \rangle$ and $UDF_E = \langle f_E^E, f_C^E \rangle$ for transforming the clusters into objects representing vertices and edges, the graph nesting operator ν_g is defined by the following procedural steps: (**i**) we must first group each vertices and edges belonging to the same cluster (a, b) with the same label through CL, and within each cluster distinguish the objects representing either entities or relationships as marked in the following α_2's containment function:*

$$\nu_1^{CL,\boldsymbol{keep}} = \alpha_{2\;\; CL,(k,s)\mapsto[k],(k,s)\mapsto[k],(k,s)\mapsto}^{\boldsymbol{keep}} \begin{bmatrix} \textbf{if}\;\; s \subseteq \phi(o.\eta, \text{“Entity”})\;\textbf{then} \\ \quad [\text{“Entity”}, s] \\ \textbf{else if}\;\; s \subseteq \phi(o.\eta, \text{“Relationship”})\;\textbf{then} \\ \quad [\text{“Relationship”}, s] \\ \textbf{else}\;\; [] \end{bmatrix}(\eta)$$

*(**ii**) then, we merge them together within the same collections, and merge them into objects by matched cluster, thus keeping entities and relationships separated:*

$$\nu_2 = \gamma^{\mathtt{tt}}_{\widetilde{GF}}(\mathtt{elect}_{\omega}(\mathtt{create}^{\omega}_{[],[],[[\text{“elements”},\phi(g,\text{“Entity”})\cup\phi(g,\text{“Relationship”})]]}(\nu_1^{CL,\boldsymbol{keep}})))$$

In particular, we have that $\widetilde{GF}$ performs this further aggregation of the nested objects by the cluster labels returned by CL as follows:

$$\widetilde{GF}(o) = \begin{cases} \ell(x) & \exists b.\ell(o) \subseteq \operatorname{cod}(CL) \\ \varnothing & \text{oth.} \end{cases}$$

*(**iii**) Then, we transform the object representing clusters either to entities (V) or to relationships (E) dependingly on the first component of the object's label. Therefore, we shall use the transformation functions:*

$$\nu_3 = \mathtt{map}_{\ell,o\mapsto \begin{cases} f_E^V(o) & \exists b.\ell(o) = [(\mathtt{e}, b)] \\ f_E^E(o) & \exists b.\ell(o) = [(\mathtt{r}, b)] \\ \xi(o) & \text{oth.} \end{cases},o\mapsto \begin{cases} f_C^V(o) & \exists b.\ell(o) = [(\mathtt{e}, b)] \\ f_C^E(o) & \exists b.\ell(o) = [(\mathtt{r}, b)] \\ \phi(o) & \text{oth.} \end{cases}}(\nu_2)$$

*(**iv**) Last, we create a new element where the clustered objects are separated by the first component of their label, representing if they are either entities or relationships as follows:*

$$\nu_g{}^{\boldsymbol{keep},\omega}_{CL,UDF_V,UDF_E}(\eta) = \mathtt{create}^{\omega}_{[],[],\begin{bmatrix} \text{“Entity”}\mapsto\left\{ x\in\varphi(o.\nu_2) \,\middle|\, x\in\phi(o.\nu_1^{CL,\boldsymbol{keep}},\text{“Entity”})\vee\exists b.\ell(x)=(\mathtt{e},b) \right\} \\ \text{“Relationship”}\mapsto\left\{ x\in\varphi(o.\nu_2) \,\middle|\, x\in\phi(o.\nu_1^{CL,\boldsymbol{keep}},\text{“Relationship”})\vee\exists b.\ell(x)=(\mathtt{r},b) \right\} \end{bmatrix}}(\nu_3)$$

◂

If we want to nest a graph by using the matched patterns as described in the introduction, we must express CL as a graph pattern matching classifier, checking whether the given vertex or edge would match the pattern, and returning the cluster identifiers to which it belongs. Please also note that we can perform pattern matching techniques over a single nested graph η as a consequence of the implementation of such languages in GSQL, as it will be described in Section 6.3.2 on page 177. Consequently, we will be able to optimize the aforementioned general approach to nesting graphs (that can be also used for graph grouping) into a more efficient operator, which only requires to generate two distinct sets of graph collections that, then, will be used to create the new nested graphs.

n-ary operators

As previously observed, the class of graph $\otimes_\vartheta$ product may be defined as a vertex join operation alongside with the creation of new edges among the returned and matched vertices by using a specific semantics, **es**. In the context of the nested graphs' edge creations, the semantics is represented by a $\langle \mathbf{es}, L, F\rangle$ triple required by the $\dot{\kappa}_{L,F}^{g'\Leftarrow \mathbf{es}}$ operator. Therefore, the graph join operation can be defined as follows:

▸ **Definition 61** ((Nested) Graph $\otimes_\vartheta$ Product)**.** *Given two nested graph operands, η and η', a ϑ predicate over the edges and an edge semantics $\langle \mathbf{es}, L, F\rangle$, such operator can be represented as follows:*

$$\eta \otimes_{JS^c,\vartheta}^{\langle \mathbf{es},L,F\rangle} \eta' = \dot{\kappa}_{L,F}^{g'\Leftarrow \mathbf{es}}(\eta \otimes_{JS^c,(x,y)\mapsto x,y\in\phi(g',\text{“Entity”})\wedge\vartheta(x,y)} \eta')$$

◂

On the other hand, the `disjoint` union (Definition 42 on page 160) clearly breaks the model constraints for nested graphs, because it doesn't preserve the containment attributes in which vertices and edges are stored. Therefore, we can use such operator only in combination with other operators that will change the structure by subsequently using a `map` operator. Therefore, instead of directly implementing it, we can continue our discussion with the set operators outlined in Subsection 6.2.1 on page 162. We observed that such operations play a double role of both object and collection operators, due to the object collection-attributes' values dualism. Nevertheless, the intersection and difference operations may also require that the edges must be always checked if may actually link some nodes appearing in a $\phi(o, \text{“Entity”})$ within the final result. The set operations over nested graphs by combining the set operations with the `constraint` as

$$\bigsqcup_{1\leq i\leq n}^{\omega} \eta_i = \texttt{constraint}\left(\bigcup_{1\leq i\leq n}^{\omega} \eta_i\right) \quad \eta\backslash\backslash^{\omega}\eta' = \texttt{constraint}(\eta\backslash^{\omega}\eta') \quad \bigsqcap_{1\leq i\leq n}^{\omega} \eta_i = \texttt{constraint}\left(\bigcap_{1\leq i\leq n}^{\omega} \eta_i\right)$$

follows: The `constraint` operator may also be adopted within the (unary) unnesting definition for nested graphs:

Graph Data Mining operators

With reference of the three world data mining model, we may observe that graph representations allow to collapse the three distinct data worlds into one. In Chapter 4 on page 89 we observed that graphs may represent both Data and Models , where – as observed in the succeeding chapters – the latter may be also generalized and interpreted as MetaModels because such models may be used as query languages.

As a consequence, the intensional world can be represented as either the graph schema (vertices' and edges' properties) or the data edges' (representing vertices' properties). Therefore, we can derive two different interpretations for the κ regionizing operator for extracting intensional data. In the first scenario, such operator is subsumed by the α_1 operator allowing the extraction of a graph schema from the given graph data. In the second scenario, κ may be interpreted as the creation of new edges between the vertices. Therefore, the link discovery operator acts as the required κ regionize operator when vertices represent data and edges represent the intensional world. Consequently, graphs are the extensional representation where each vertex is connected to its edges.

The mining loop λ operator may be defined via the higher order `fold` operator, which allows to perform iterations over arbitrary data collections.

Given that in the second scenario graphs already provide the extensional world, we now have to discuss Pop for the first scenario. Such operator must associate each data representation to the schema by extending the intensional representation of a graph schema (or pattern) P with the data it matches. Each object o in P is associated to the objects $f_j(o)$ in η resulting from the morphism f_j generated by $m_P(\eta)$.

$$Pop_m(\eta, P) = \varepsilon_{o \mapsto \bigoplus_{f_j \in m_P(\eta)} [[\text{“j”}, f_j(o)]]}(P) \tag{6.8}$$

Nested graphs (and more generally GSMs) provide the required extensional representation for Pop operator. If we unnest the extensional information we may obtain the original data values with its original schema ($\pi_A \overset{\text{def}}{=} \mu^{\mathsf{ff}}_{\mathsf{tt}}$), while the π_{RDA} operation may forget the nested information may be defined by a simple map operation which forgets all the morphism-nested information within the nested graph and retains the objects that have been actually matched in the previous phase, thus acting as projection operator.

Finally, this section showed that the joint combination of the nested graph model and GSQL provide the characterization of a data mining model for graphs via nested graphs, where an uniform set of both data representations and operators is given.

6.3.2 Implementing traversal query languages' semantics (σ)

Please refer to Section 3.5.1 on page 77 for the graph query language terminology adopted in this subsection.

Both graph selection and graphs extraction query languages rely on visiting the input data graph and then returning either the visited part (NautiLOD [FPG15]) or the data that is reached after evaluating the visiting steps (XPath [BBC⁺15]). If we consider trees as a specific graph, both languages are (graphs) extraction languages, even though they substantially differ on how both traversing is performed and on the returned results: while NautiLOD returns a graph which subgraphs match a given specification, XPath returns a forest of trees that can be reached after the traversal process. These two languages may be also distinguished from the way they traverse the GSM data structure: while the first performs a navigation of the graph data structure by alternatively moving across objects belonging to different containments (vertices and edges) and does not necessarily involve a visit of the GSM by using φ^* recursively, the second query language visits in depth (φ) all the objects belonging to the same root object and performs a visit in depth.

A combination of these two traversal approaches leads to a complete nested graph traversal language. Even though this thesis is not going to provide a user friendly syntax for such query language, it is going to provide an example on how to interpret such languages into GSQL, so that it can be used to navigate GSM data structures. Then, given that both languages may be represented in GSQL (and hence, at the semantics level), this means that such languages may be also freely integrated even at a syntactic level. The definition of another nested graph traversing language combining both features then comes for granted. The final language may provide the selection predicate, which is defined as follows:

▸ **Definition 62** (Selection). *Given a GSM $\eta = (g, O, \ell, \xi, \phi)$, a traversal algorithm s and a pattern P, the selection operator $\sigma_P(\eta)$ returns a GSM which is a substructure of η through the execution of the query P on η interpreted with s, where $s(P)$ is interpreted as a sequence of GSQL operators $(s(P)(\eta))$. Consequently:*

$$\sigma_{s,P}(\eta) = (s(P))(\eta) = (g', O', \ell', \xi', \phi')$$

In particular each semantics "s" must guarantee to return a substructure, compliant to the constraints of the original input data. E.g., for nested graph we must ensure that $\phi(\mathfrak{g}', \text{“Entity”}) \subseteq \phi(\mathfrak{g}, \text{“Entity”})$, $\phi(\mathfrak{g}', \text{“Relationship”}) \subseteq \phi(\mathfrak{g}, \text{“Relationship”})$ *and* $O' \subseteq O$*, where each edge* i *in* e' *has source and target it* v' $(\forall i \in \phi(e').\lambda(e') \in \phi(v') \times \phi(v'))$. ◂

In the following subsections we're going to show how to define such s expressions for translating traversal query languages into GSQL expressions.

XPath Traversals

With reference to Definition 10 on page 68 where we outlined a minimal subset of XPath, we now focus on the path traversal discarding all the axis notation and the selection predicates. This core language provides the minimal language allowing to traverse data structures in depth. The reason of doing so is that we want to use such language only for traversing the data structure in depth, and not to filter the obtained data, which can be easily achieved in our algebra via `map`.

Moreover, on the footsteps of [FPG15], which did a similar approach for graph traversals, we're going to show that is possible to express a nested data structure traversal using algebraic operators. In particular, we're going to show that we can define the semantics $XPathInit$ interpreting the minimal XPath expressions.

▸ **Definition 63** (Minimal XPath (GSQL Semantics)). *$XPathInit$ interprets the Minimal XPath query over a GSM representation of XML documents using GSQL. Before evaluating the* `<`μ`XPath>` *expression* P *via XPath, such function preventively creates a new object with a fresh id* ω *where the result is going to be stored in* $\phi(\omega, \text{“result”})$*. Therefore:*

$$XPathInit(P) = n \mapsto XPath(P)(\texttt{elect}_\omega(\texttt{create}^{\omega}_{[],[],[\![\{\texttt{``result''},[g]\}]\!]_\Gamma}(n)))$$

At this point, we can define the semantics associated to the expression as follows:

$$XPath(P) = n \mapsto \begin{cases} \texttt{map}_{\ell,\xi,\phi\oplus\{\omega,\{\texttt{``result''},t(cSem(\texttt{<c>}))\}\}}(n) & P \equiv \texttt{<c>}\star \\ \texttt{map}_{\ell,\xi,\phi\oplus\{\omega,\{\texttt{``result''},\ t\left(\texttt{select}(cSem(\texttt{<c>})\ :\ \texttt{y -> \{<s> in y.}\mathbf{\texttt{ell}}\texttt{\}})\right)\}\}}(n) & P \equiv \texttt{<c><s>} \\ XPath(\texttt{<}\mu\texttt{XPath>})(XPath(\texttt{<c><s>})n) & P \equiv \texttt{<c><s><}\mu\texttt{XPath>} \end{cases}$$

where the `union` *of the* `script` *lists can be defined as* `union = x ->distinct {x[0] ++x[1]}`*, and the interpretation of the subSelectors is defined as follows:*

$$sSem(u,s) = \begin{cases} \texttt{u[s]} & s \neq \star \wedge s \neq \varepsilon \\ \texttt{foldl \{[], map(u : y -> y[1]), union\}} & \text{oth.} \end{cases}$$

$$cSem(\texttt{<c>}) = \begin{cases} sSem(\texttt{x[0].}\mathbf{\texttt{phi}},\texttt{<s>}) & \texttt{<c>} \equiv /^{\texttt{<s>}} \\ sSem(\texttt{x[0].}\mathbf{\texttt{varphiplus}},\star) & \texttt{<c>} \equiv //^{\star} \end{cases}$$

$$t(expr) = \texttt{foldl \{[], g.}\mathbf{\texttt{phi}}\texttt{[``result''], x} \mapsto \texttt{\{}expr \texttt{ ++ x[1]\}\}}$$

◂

Consequently, instead of using many distinct algebraic operator for performing a traversal of the nested structure as in [MM06], we may now traverse the whole data structure by providing the full XPath expression, and then performing the desired operation over the forest of selected substructures.

▸ **Example 31.** *As an example, let us try to express the XPath traversal query:* `/medical/patient/name`. *Given that τ_{XML} converts each tag into the "Tag" containment, we can rewrite the query as* $/^{\text{``Tag''}}\mathtt{medical}/^{\text{``Tag''}}\mathtt{patient}/^{\text{``Tag''}}\mathtt{name}$. *By the definition of the aforementioned GSQL semantics for XPath, from* $\mathit{XPathInit}(/^{\text{``Tag''}}\mathtt{medical}/^{\text{``Tag''}}\mathtt{patient}/^{\text{``Tag''}}\mathtt{name})$ *we obtain the following expression:*

$$\mathit{XPathInit}(P)(n) = \mathit{XPath}(/^{\text{``Tag''}}\mathtt{medical}/^{\text{``Tag''}}\mathtt{patient}/^{\text{``Tag''}}\mathtt{name})(\mathtt{elect}_{\omega}(\mathtt{create}^{\omega}_{[],[],[\![\{\texttt{``result''},[g]\}]\!]_{\Gamma}}(n)))$$

$$\begin{aligned}
&\mathit{XPath}(/^{\text{``Tag''}}\mathtt{medical}/^{\text{``Tag''}}\mathtt{patient}/^{\text{``Tag''}}\mathtt{name})(\mathtt{elect}_{\omega}(\mathtt{create}^{\omega}_{[],[],[\![\{\texttt{``result''},[g]\}]\!]_{\Gamma}}(n))) =\\
&\qquad \mathit{XPath}(/^{\text{``Tag''}}\mathtt{patient}/^{\text{``Tag''}}\mathtt{name})(\mathit{XPath}(/^{\text{``Tag''}}\mathtt{medical})(\mathtt{elect}_{\omega}(\mathtt{create}^{\omega}_{[],[],[\![\{\texttt{``result''},[g]\}]\!]_{\Gamma}}(n)))) =\\
&\qquad \mathit{XPath}(/^{\text{``Tag''}}\mathtt{name})(\mathit{XPath}(/^{\text{``Tag''}}\mathtt{patient}(\mathit{XPath}(/^{\text{``Tag''}}\mathtt{medical})(\mathtt{elect}_{\omega}(\mathtt{create}^{\omega}_{[],[],[\![\{\texttt{``result''},[g]\}]\!]_{\Gamma}}(n)))))) =
\end{aligned}$$

In particular, each internal expression designing one part of the whole visiting step can be rewritten for `medical` *as follows:*

$$\begin{aligned}
\mathit{XPath}(/^{\text{``Tag''}}\mathtt{medical})(n) &= \mathtt{map}_{\ell,\xi,\phi\oplus\{\omega,\{\texttt{``result''},\ t\left(\texttt{select}(\mathit{cSem}(/^{\text{``Tag''}})\ :\ \texttt{y -> \{``medical'' in y.}\mathbf{ell}\texttt{\})}\right)\}\}}(n)\\
&= \mathtt{map}_{\ell,\xi,\phi\oplus\{\omega,\{\texttt{``result''},\ t\left(\texttt{select}(\mathit{sSem}(\texttt{x[0].}\mathbf{phi},\text{``Tag''})\ :\ \texttt{y -> \{``medical'' in y.}\mathbf{ell}\texttt{\})}\right)\}\}}(n)\\
&= \mathtt{map}_{\ell,\xi,\phi\oplus\{\omega,\{\texttt{``result''},\ t\left(\texttt{select(x[0].}\mathbf{phi}[\text{``Tag''}]\ :\ \texttt{y -> \{``medical'' in y.}\mathbf{ell}\texttt{\})}\right)\}\}}(n)\\
&=
\end{aligned}$$

$$\mathtt{map}_{\ell,\xi,\phi\oplus\{\omega,\{\texttt{``result''},\ \texttt{foldl \{[], g.}\mathbf{phi}\texttt{[``result'']},\ \texttt{x}\mapsto\texttt{\{select(x[0].}\mathbf{phi}[\text{``Tag''}]\ :\ \texttt{y -> \{``name'' in y.}\mathbf{ell}\texttt{\}) ++ x[1]\}\}\}\}}}(n)$$

Please note that the present semantics does not provide one substructure of the GSM, while provides a collection of objects which are effectively substructures of the original GSM.

NautiLOD Traversals

In Table 3.4 on page 79 we provided the `Successful` NautiLOD's semantics over MPGs. This paragraph shows that we can express such graph traversal semantics in GSQL after mapping MPGs and MPGs collections into GSM nested graphs. In particular, we use ℓ to store the starting node of the visit, and ξ for storing the ending nodes' collection.

The first class of operators are the ones over the graphs, where only the concatenation $\circ$ and a distinct union for MPGs[5] $\sqcup$ and MPGs' collections[6]. The first concatenation operator can be defined as follows:

▸ **Definition 64** (NautiLOD concatenation). *Given two nested graphs η_L and η_R representing MPG pointed graphs, we can define the NautiLOD concatenation for GSM Pointed Graphs as follows:*

[5] In the original paper, such operator is defined as $\cup$. Given that we have already used $\sqcup$ for our operator over GSMs and given that their operator also uses the

[6] In the original paper, such collection operator is defined as $\oplus$. We prefer to use the $\cup$ notation as the one originally addressed for GSMs, and hence, for labelled collections.

$$\phi_\circ = \left[\ \omega \mapsto \begin{cases} \phi(\omega) & \ell(o.\eta_R) \subseteq \xi(o.\eta_L) \\ \varnothing & \text{oth.} \end{cases}\ \right]$$

$$\ell_\circ = \left[\ \omega \mapsto \begin{cases} \ell(o.\eta_L) & \ell(o.\eta_R) \subseteq \xi(o.\eta_L) \\ \varnothing & \text{oth.} \end{cases}\ \right] \qquad \xi_\circ = \left[\ \omega \mapsto \begin{cases} \xi(o.\eta_R) & \ell(o.\eta_R) \subseteq \xi(o.\eta_L) \\ \varnothing & \text{oth.} \end{cases}\ \right]$$

$$\eta_L \circ \eta_R = \mathtt{map}_{\ell\oplus\ell_\circ,\ \xi\oplus\xi_\circ,\ \phi\oplus\phi_\circ}(\eta_L \cup^\omega \eta_R)$$

◀

On the other hand, the NautiLOD union of matched subgraph patterns can be defined as follows:

▸ **Definition 65** (NautiLOD union). *Given two nested graphs η_L and η_R representing MPG pointed graphs, we can define the NautiLOD union for GSM Pointed Graphs as follows:*

$$\phi_\sqcup = \left[\ \omega \mapsto \begin{cases} \phi(\omega) & \ell(o.\eta_R) = \ell(o.\eta_L) \\ \phi(o.\eta_L) & \phi(o.\eta_R, \text{“Entity”}) = \phi(o.\eta_R, \text{“Relationship”}) = \varnothing \\ \phi(o.\eta_R) & \phi(o.\eta_L, \text{“Entity”}) = \phi(o.\eta_L, \text{“Relationship”}) = \varnothing \\ \varnothing & \text{oth.} \end{cases}\ \right]$$

$$\ell_\sqcup = \left[\ \omega \mapsto \begin{cases} \ell(o.\eta_L) & \ell(o.\eta_R) = \ell(o.\eta_L) \\ \ell(o.\eta_L) & \phi(o.\eta_R, \text{“Entity”}) = \phi(o.\eta_R, \text{“Relationship”}) = \varnothing \\ \ell(o.\eta_R) & \phi(o.\eta_L, \text{“Entity”}) = \phi(o.\eta_L, \text{“Relationship”}) = \varnothing \\ \varnothing & \text{oth.} \end{cases}\ \right]$$

$$\xi_\sqcup = \left[\ \omega \mapsto \begin{cases} \xi(o.\eta_R) \cup \xi(o.\eta_L) & \ell(o.\eta_R) = \ell(o.\eta_L) \\ \xi(o.\eta_L) & \phi(o.\eta_R, \text{“Entity”}) = \phi(o.\eta_R, \text{“Relationship”}) = \varnothing \\ \xi(o.\eta_R) & \phi(o.\eta_L, \text{“Entity”}) = \phi(o.\eta_L, \text{“Relationship”}) = \varnothing \\ \varnothing & \text{oth.} \end{cases}\ \right]$$

$$\eta_L \sqcup^\omega \eta_R = \mathtt{map}_{\ell\oplus\ell_\sqcup,\ \xi\oplus\xi_\sqcup,\ \phi\oplus\phi_\sqcup}(\eta_L \cup^\omega \eta_R)$$

◀

Given that $\cup$ (originally $\oplus$) denotes the union between GSMs collections, such operation can be directly expressed through the $\bigcup$ GSQL operator. Please also note that in the original paper is also relevant to extend the two previously-mentioned operations to graph collections, such that those operators can be applied to all the graphs appearing in the resulting collections. For this reason we can choose to instantiate the $\otimes\theta$-Product by replacing $\otimes$ with one of the aforementioned operators and by choosing a θ predicate that can be eventually the always true predicate. Please also note that the resulting collection must be re-labelled at the end of the step in order to achieve an uniform notation.

▸ **Example 32.** *Suppose that we now want to traverse a graph represented as a nested graph η from any given initial paper u, of which we want to know its authors' affiliations. NautiLOD expressions, similar to XPaths', represent such graph traversal query as follows:*

```
authoredBy/affiliatedTo
```

Among all the possible semantics proposed by NautiLOD, we adopt in this example the Successful *(S) over our nested (data) graph* η*. In particular, given that NautiLOD's action expressions are not relevant for our graph query purposes because they provide customizable side effects not affecting the final result, we may remove them from the query interpretation. Therefore, the interpretation of such semantics reduces to the rewriting of the NautiLOD expression:*

$$\begin{aligned}
[\![\cdot\texttt{authoredBy/affiliatedTo}]\!](u) &= \bigcup [\![\texttt{authoredBy/affiliatedTo}]\!](u)\\
&= \bigcup\left([\![\texttt{authoredBy}]\!](u) \circ \left(\bigcup_{v\in T_\Gamma,\Gamma\in[\![\texttt{authoredBy}]\!](u)} [\![\texttt{affiliatedTo}]\!](v)\right)\right)\\
&= \bigcup_{v\in T_\Gamma,\Gamma\in[\![\texttt{authoredBy}]\!](u)} \Gamma\circ[\![\texttt{affiliatedTo}]\!](v)
\end{aligned} \tag{6.9}$$

We use the `fold` *iteration over the content of a given* η' *GSM for defining the recursive application of the operator on all the elements of the set. Therefore, we use* bigop *to apply recursively operation* op *to a GSM-elected object identified by* ς*, which accepts all the arguments of the folding function:*

$$\mathrm{bigop}_{\mathsf{op},\varsigma}(\eta') \overset{\text{def}}{=} \texttt{fold}_{\{\,(o_k,v)\mid o_k\in\phi(\omega_c,\text{“Entity”}),v\in\xi(o_k)\,\},((o_k,v),\alpha)\mapsto(\texttt{elect}_{o_k}(\alpha)\;\mathsf{op}\;\varsigma(o_k,v,\alpha))}(\eta')$$

At this point, we have to define how to extract each graph containing a *as a label, originating from* u *(*$[\![\mathsf{a}]\!](u)$*). This solution can be achieved by using the graph nesting operator, through which we create a nested graph containing only nested vertices, which contain a subgraph of the initial dataset:*

$$\begin{aligned}
&[\![\mathsf{a}]\!](u) \overset{\text{def}}{=}\\
&\overset{\text{def}}{=} \mu^{\mathtt{ff}}_{\mathtt{tt}}(\\
&\qquad \mathtt{map}_{o\mapsto\begin{cases}\phi(x,\text{“src”}) & o\in\phi(g,\text{“Entity”}),\ell(o)=[(\mathsf{v},x)]\\ \varnothing & \text{oth.}\end{cases},\ell,\xi}\left(\nu^{\mathtt{ff},\omega}_{g_{CL^u_a,UDF^{a,u}_V,UDF^{a,u}_E}}(\eta)\right)\\
&\quad)
\end{aligned}$$

Such definition uses the following functions:

$$S^u_a = \{\,(\mathsf{v},x)\mid x\in\phi(g,\text{“Relationship”})\wedge a\in\ell(x)\wedge u\in\phi(x,\text{“src”})\,\}$$

$$CL^u_a(x) = \begin{cases}\{\,(\mathsf{v},y)\in S^u_a\mid x\in\phi(y,\text{“src”})\cup\phi(y,\text{“dst”})\,\} & x\in\phi(g,\text{“Entity”})\\ (\mathsf{v},x) & (\mathsf{v},x)\in S^u_a\end{cases}$$

$$UDF^{a,u}_E = \langle[[\,]],\phi\rangle \qquad UDF^{a,u}_V = \left\langle o\mapsto\begin{cases}\phi(x,\text{“dst”}) & \ell(o)=[(\mathsf{v},x)]\\ \varnothing & \text{oth.}\end{cases},\phi\right\rangle$$

After the previous definitions, we can finally write the GSQL semantics as:

$$\mu^{\mathtt{ff}}_{\mathtt{tt}}(\mathrm{bigop}_{\circ,(\_,v,\_)\mapsto[\![\texttt{affiliatedTo}]\!](v)}([\![\texttt{authoredBy}]\!]^{\omega_c}(u)))$$

Similar considerations may be provided for the whole set of NautiLOD syntax, which do not fall within the aims of the present thesis. Nevertheless, we showed that it is possible to provide a graph traversal semantics on GSM which are closed under the GSQL algebraic operators used to represent NautiLOD's semantics.

6.3.3 Representing *is-a* aggregations

Within Section 2.2.2 on page 40 and specifically on Examples 6 and 7, we have already touched upon structural graph aggregation. We also observed that aggregations on top of graph data structures do not allow drill-down operations on top of aggregated data. Even if the formal operator performing such operation on top of nested graphs will be fully provided in Chapter 7 on page 195, in the following two subsections we will show how GSM allows to integrate tree (hierarchy) and relational data extracted from a graph, so that it can be drill-down in a second step by the user.

As we saw in Example 8 on page 44 where data aggregation operations were briefly introduced, there is a need for representing nested components to which an aggregated representation is provided. In particular, GSMs can explicitly associate to each object o a collection $\phi(o,p)$ of elements over which aggregation functions p can be evaluated over the data in such collection.

▸ **Example 8** (continuing from p. 44). *We want to show how is-a aggregations are possible by using already existing data hierarchies. Since our data model violates the first normal form - similarly to other semi-structured data models - it is possible to express the hierarchy in Figure 2.18a on page 45 through the following instantiation:*

$$H=(r,\{\,r,t_1,t_2,c_1,\dots,c_4,p_1,\dots,p_8\,\},\ell,\xi,\phi)$$

The reference object r represents the root of the hierarchy and is defined as follows:

$$\ell(r)=[\texttt{root}]\quad \phi(r,\texttt{parentof})=[t_1,t_2]$$

In particular, each element of the hierarchy is defined as follows:

$$\ell(t_1)=[\texttt{type}]\quad \xi(t_1)=[\text{“House cleaner”}]\quad \phi(t_1,\text{“parentof”})=[3,4]$$

$$\ell(t_2)=[\texttt{type}]\quad \xi(t_2)=[\text{“Food”}]\quad \phi(t_2,\text{“parentof”})=[5,6]$$

$$\ell(c_1)=[\texttt{category}]\quad \xi(c_1)=[\text{“Cleaner”}]\quad \phi(c_1,\text{“parentof”})=[7,8]$$

$$\ell(c_2)=[\texttt{category}]\quad \xi(c_2)=[\text{“Soap”}]\quad \phi(c_2,\text{“parentof”})=[9,10]$$

$$\ell(c_3)=[\texttt{category}]\quad \xi(c_3)=[\text{“Diary Product”}]\quad \phi(c_3,\text{“parentof”})=[11,12]$$

$$\ell(c_4)=[\texttt{category}]\quad \xi(c_4)=[\text{“Drink”}]\quad \phi(c_4,\text{“parentof”})=[13,14]$$

$$\ell(p_1)=[\texttt{product}]\quad \xi(p_1)=[\text{“Shiny”}]\qquad \ell(p_2)=[\texttt{product}]\quad \xi(p_2)=[\text{“Brighty”}]$$

$$\ell(p_3)=[\texttt{product}]\quad \xi(p_3)=[\text{“CleanHand”}]\qquad \ell(p_4)=[\texttt{product}]\quad \xi(p_4)=[\text{“Marseille”}]$$

$$\ell(p_5)=[\texttt{product}]\quad \xi(p_5)=[\text{“Milk”}]\qquad \ell(p_6)=[\texttt{product}]\quad \xi(p_6)=[\text{“Yogurt”}]$$

$$\ell(p_7)=[\texttt{product}]\quad \xi(p_7)=[\text{“Water”}]\qquad \ell(p_8)=[\texttt{product}]\quad \xi(p_8)=[\text{“Coffee”}]$$

We now want to use this hierarchy as a dimension for the `Product` *objects in Figure 2.14 on page 38. As a start, let us suppose that each* `Product` *vertex is associated to a* `quantity` *attribute, providing the sum of the* `quantity` *of the ingoing* `item` *edges. Since such aggregation functions can be only expressed over the data collections and given that there are no particular constraints on the types of the ids to be stored in such data collections, we have that each expression shall provide the following aggregation function ($\xi'[0]$):*

```
quantity := (foldl {0,
                    o.phi ["quantity"],
                    x -> { (foldl {0,
                                  map(select(x[0].phi["Attribute"] : y -> {y.ell == "quantity"
                                      ↪ }))}) }
                                      : y -> { y.xi[0] } ),
                                  y -> { y[0] + y[1] }
                                 }) }
                   }
            )
```

In this case we can use ξ to store the definition of `quantity` *function, that will differ from the one used within hierarchies. This other function is going to be defined as follows ($\xi''[0]$):*

```
        quantity := (foldl {0, o.phi ["quantity"], y -> {(y[0].xi[0] x[0])+y[1]}})
```

In this way, the functions' evaluation may be evaluated directly by traversing the hierarchy, without any required need of effectively aggregating the data. As a next step, from the whole company database we can select only the products (filtering through σ), and then extend each of them with a new `quantity` *attribute aggregating their incoming edges' quantity values. Such statement can be expressed through the following statement:*

$$P' = \texttt{filter}_{x\mapsto\text{“Product”}\in\ell(x)}\Big($$
$$\texttt{map}_{\ell,\xi',\phi\oplus_v[[v,[\text{“quantity”},[v\in\phi(g_{DB},\text{“Relationship”})|\exists u.\lambda(e)=(u,v)\land\text{“Product”}\in\ell(v)]]]]}(DB)$$
$$\Big)$$

At this point, we want to do the same operation for the hierarchy: we must first rename `parentof` *into* $\texttt{quantity}(x)$*, and then just replace the* `parentof` *placeholder with the function performing the aggregation:*

$$H' = \texttt{map}_{\ell,\xi'',\phi\oplus_s[[s,[\text{“quantity”},\phi_H(s,\text{“parentof”})]]]}(H)$$

In order to be able to evaluate the aggregation at different hierarchy's abstraction levels, discard all the other informations from the database's products and keep only the given dimension, we can finally perform the following left join (that can be defined from the $\otimes_\theta$ product):

$$H' ⟕ \pi_{\texttt{name},\texttt{quantity}}(P')$$

Please note that this final join was possible because the products within the hierarchy do not have a `parentof` *field, and hence they have no* `quantity` *one. At this point, the amount of the products bought by the company's employees can be done by evaluating the* $\texttt{quantity}(x)$ *expression over its nested content: the expression's evaluation will recursively visit the remaining part of the hierarchy joined with the data.*

6.3.4 Generalized Graph Grammars $\mathcal{G}$ for Nested Graphs. $\varrho^{\mathcal{G}}_{\mathcal{H},\mathcal{T}}(\mathcal{H})(\eta)$

We continue to both analyse the dependency graph semistructured representation for unstructured data (as already introduced in Section 2.1.4 on page 28), and search for a ϱ definition. As also remarked in Section 3.3 on page 70, the problem with representing unstructured data is that they lack of a precise data structure that could be directly queried, handled and created by automated processes. As a consequence, some "structure" has to be provided.

If we suppose that the full text is well written and that it follows the correct grammar rules of its natural language, then we can express such full text as a dependency graph (Figure 6.4 on page 185), and we can express grammatical rules through graph grammar operations. In particular, we can use graph pattern matching queries to match all the foreseeable natural language's constructs, and then transform them into a nested graph, where entities will represent whole *complements* or whole sentences (e.g., *dependent or subordinate clauses*), and verbs - representing actions from the subject to the object or from the agent towards the subject - represent edges connecting the former concepts through linguistic relationships.

On the footsteps of GraphLOG, Figure 6.5a provides an example of a visual representation of match ($\mathcal{H}$) and rewrite ($\mathcal{T}$) rules: we want to match the graphs represented in the "pattern" column, and rewrite them as the nested graphs presented within the "result" column. The first rule tells that each vertex X that has outgoing edges with one of the labels depicted in λ has to be updated with the $value$ stored in its correspondent adjacent vertex Y. The second rule tells to collect inside one single nested vertex all the elements that are in conjunction $conj$ with each other through a conjunction Z: moreover, any other ingoing and outgoing edge to this nested object of a given label must be rewritten as a single ingoing or outgoing edge. The last one tells to rewrite the verb of a sentence, represented as a root V linked to the list of subjects $\vec{S}$ and the object $X||V$ (that can be a verb being a root of another sentence), as a whole nested vertex containing an edge to the nested object of all the components of the subject, directed to the final object or sentence. In particular, graph grammars are defined through the combination of such rewriting rules [CM90b, Plu99]: the former picture also shows how such match and rewrite query can be expressed by a visual formalism, instead of being coded using a specific programming language as already showed in the early 90s [CM90b].

A possible outcome of such pattern matching and transformation is provided in Figure 6.5b: this result is both more human readable and machine readable than the first sentence input, thus allowing a computer to better analyse the content of the sentence. As we have already seen in Section 3.5.2 on page 79, graph grammars neither allow to nest graphs nor allow to fully transform the matched vertices and edges. Consequently, we have to generalize such approach in this regard. Moreover, we also have to allow the transformation of multiple pattern graphs into one single rewriting graph (hub schema). The following definition provides a preliminary and wordy definition, that is going to be formalized after an exaplainatory running example on a much simpler use case involving bibliography graphs.

▸ **Definition 66** (Generalized Graph Grammars). *A **generalized graph grammar**[7] is a collection of n graph rewriting rules $[H_i \to T_i]^{m^i}_{i\leq n}$ applied to nested graph ɳ via the following operator $\mathcal{G}^{m_1,\dots,m_n}_{[H_i\to T_i]_{i\leq n}}$(ɳ), returning another nested graph. In particular, the set of all the H_is appearing in the rules is denoted by $\mathcal{H}$, while the T_is are referenced by $\mathcal{T}$.*

The associated semantics of such graph grammar on a given nested graph in input ɳ is the following: each graph head H_i provides a graph pattern matching query which is going to be interpreted through a semantics m^i returning a collection $m^i_{H_i}$(ɳ) of morphisms f_j as a result of the matching phase.

*The associated transformation T_i is described through a nested graph, and it is instantiated on each generated morphism f_j generating several graphs replacing the matched ones. Each transformation graph T_i may contain **(i)** vertices (and/or edges) o_h also contained[8] in H_i providing*

[7] See Sherlock for both a representation of nested graphs and an implementation of such generalized graph grammars: `http://rebrand.ly/unibo7252c`

[8] We preferred to use object identity instead of using further morphisms as in both graph grammars

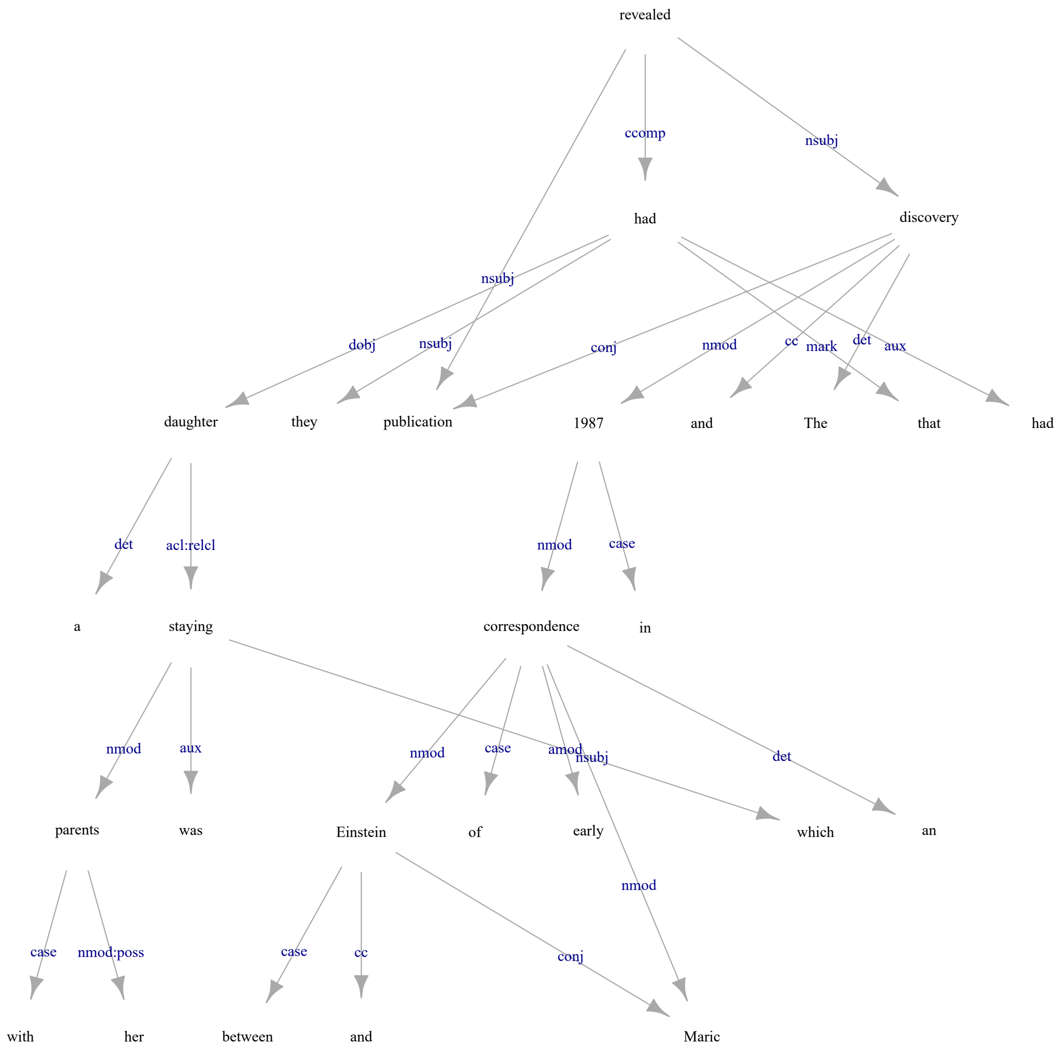
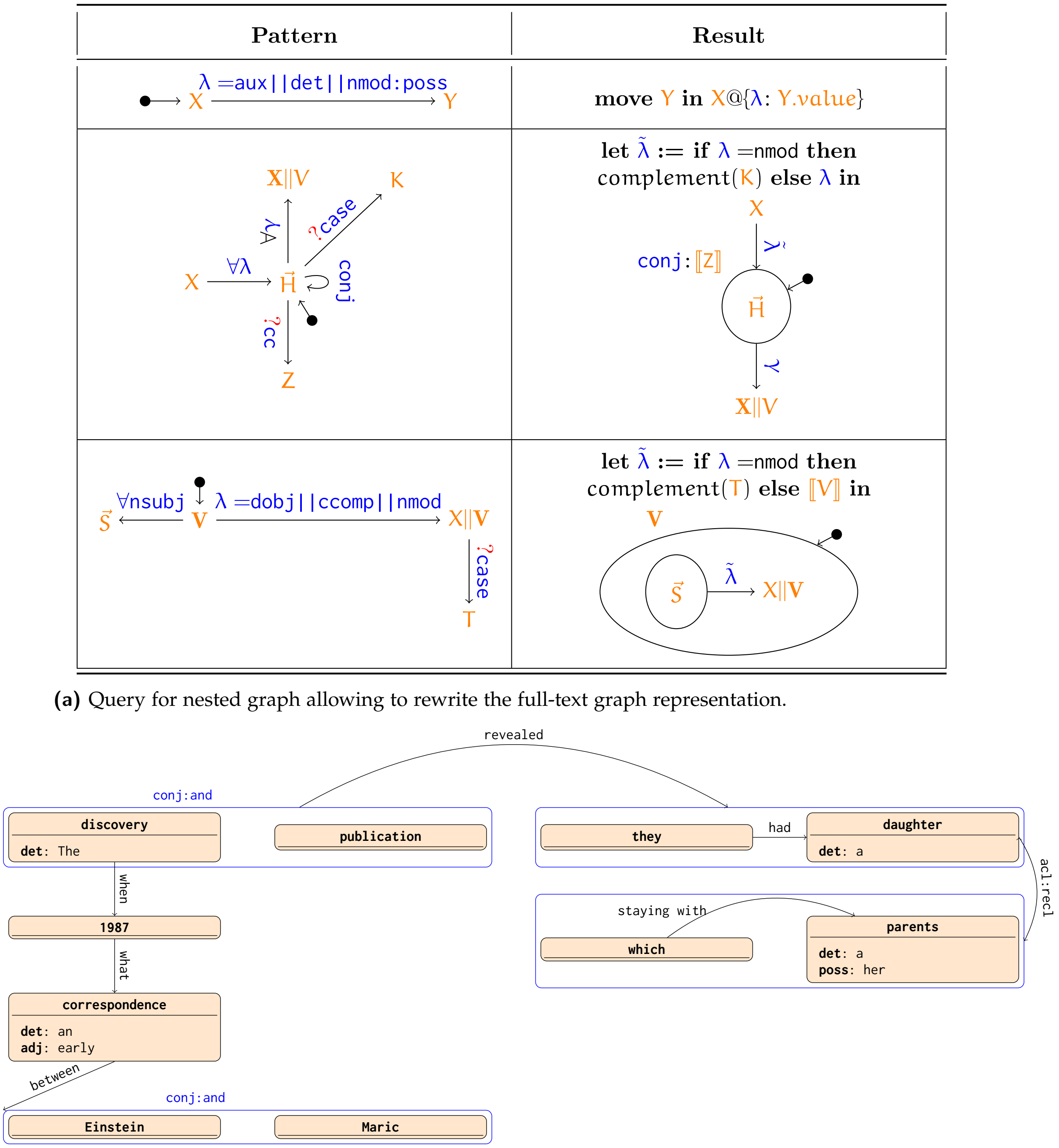


Figure 6.4 Graph representation of the following sentence from Wikipedia: *"The discovery and publication in 1987 of an early correspondence between Einstein and Maric revealed that they had had a daughter was staying with her parents."*.



(a) Query for nested graph allowing to rewrite the full-text graph representation.

(b) Result of the application of the patterns in Figure a.

Figure 6.5 Translating the full-text representation as a graph in Figure 6.4: this requires a graph transformation language and the definition of nested graphs.

the semantics "return all the matched vertices (and/or edges) by a given vertex (and/or predicate) represented by o_h" and possibly apply a transformation, ***(ii)*** *or new vertices (and/or edges) o_c that are not in the vertex set $\phi(g, \text{"Entity"})$, thus providing the semantics "transform (or aggregate) the matched vertices (and/or edges) according to the (aggregation) functions of o_c". Consequently,* ***(iii)*** *all the vertices (and edges) that are matched in H_i but not returned as vertices in* ***(i)*** *(or edges linking vertices that are neither contained in other objects nor appear as vertices in the final result) are removed from the returned graph. In particular, all the edges generated in* ***(ii)*** *must be associated with all the returned or generated vertices by the pattern.* ◂

The following example motivates the operator's subsequent formalization.

▸ **Example 33.** *With reference to Example 7 on page 41, we want to show how such aggregation is possible by simply using graph grammar rules. In order to obtain the same result in Figure 2.17b on page 44, we can use the Generalized Graph Grammars with just one rule $\mathcal{G} = [H \to T]^s$, where H is the query "return all the authors that are co-authors" nested graph:*

$$H = (h, \{ h, u_0, u_1, p_0, e_0, e_1 \}, \ell, \xi, \phi)$$

$$\ell(u_0) = \{\texttt{Author}\} = \ell(u_1) \qquad \ell(p_0) = \{\texttt{Paper}\} \qquad \ell(e_1) = \{\texttt{AuthorOf}\} = \ell(e_2)$$

$$\xi(u_0) = [\text{"isGroupBy"}, \text{"}\texttt{o.id != } u_1\text{"}]$$

$$\xi(u_1) = [\text{"isGroupBy"}, \text{"}\texttt{o.id != } u_0\text{"}]$$

$$\phi(h, \text{"Entity"}) = [u_0, u_1, p_0] \qquad \phi(h, \text{"Relationship"}) = [e_0, e_1]$$

$$\phi(e_0, \text{"src"}) = [u_0] \quad \phi(e_0, \text{"dst"}) = [p_0] \quad \phi(e_1, \text{"src"}) = [u_1] \quad \phi(e_1, \text{"dst"}) = [p_0]$$

This query then expresses that the pattern matching query must return the pattern fixing the two user ("isGroupBy"), and let return all the papers in common, and not just one single paper. We can enforce the fact that u_0 and u_1 cannot be the same user within the graph, if we allow vertices to insert predicates within their expressions as the second condition provided above.

The T graph states that only the authors must be returned and that a new edge between the co-authors must be created. Moreover, all the papers must be removed from the final result. In particular, the edge can contain all the papers that will be matched by p_0 as follows:

$$(t, \{ t, u_0, u_1, ca_0, p_0 \}, \ell, \xi, \phi)$$

$$\xi(ca_0) = [\text{"}\texttt{o.src!= o.dst}\text{"}]$$

$$\ell(u_0) = \{\texttt{Author}\} = \ell(u_1) \qquad \ell(ca_0) = \{\texttt{coauthorship}\}$$

$$\phi(t, \text{"Entity"}) = [u_0, u_1] \quad \phi(t, \text{"Relationship"}) = [ca_0]$$

$$\phi(ca_0, \text{"src"}) = [u_0] \quad \phi(ca_0, \text{"dst"}) = [u_1] \quad \phi(ca_0, \text{"pp"}) = [p_0]$$

Since the grammar rules can only tell how to transform the matched subgraphs, we must first restrict the bibliography network BN in Figure 2.17a on page 44 into the greatest subgraph matching H: therefore, we're going to apply our graph grammar rules over $BN' = \sigma_{s,H}(BN)$. As a consequence, BN' will not contain the `Paper` *with id 5, because was only authored by the* `Author` *with id 1, and hence is not involved in any coauthorship relation.* ◂

and nested graph alignments' refinement as in the previous chapter. Transformations and embeddings are going to be expressed as target objects' containments.

Before providing the formal definition of the semantics of such operator, we shall provide some explanations on how to generate the final formula, step by step. In particular, the removal of vertices in **(iii)** from the vertex set of the result can be defined as follows:

$$\delta V = \phi(\mathfrak{g},\text{“Entity”})\backslash\texttt{fold}_{\mathcal{G},\ ([H_i\to T_i]^{m_i},\alpha)\mapsto\texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto f_j\left(\phi(\mathfrak{g}_i^H,\text{“Entity”})\backslash\phi(\mathfrak{g}_i^T,\text{“Entity”})\right)\cup\beta}(\alpha)}([]) \tag{6.10}$$

This expression has to be read as follows: given a nested graph η, from its vertex set remove all those vertices that appear in the morphism f_j originated by a matching o_h that is not represented in T_i. This is performed , for each graph pattern H_i belonging to the i-th rule of the grammar associated with a match semantics m^i over which morphisms f_j are provided,

▸ **Example 33** (continuing from p. 187). *Each morphism f_j returned from the match of BN' with H will have as many keys as the vertices and edges in H. In particular we will have four morphisms:*

$$f_0(u_0) = [0] \quad f_0(u_1) = [2] \quad f_0(p_0) = [3] \quad f_0(e_0) = [6] \quad f_0(e_1) = [7]$$

$$f_1(u_0) = [2] \quad f_0(u_1) = [0] \quad f_0(p_0) = [3] \quad f_0(e_0) = [7] \quad f_0(e_1) = [6]$$

$$f_2(u_0) = [2] \quad f_0(u_1) = [1] \quad f_0(p_0) = [4] \quad f_0(e_0) = [8] \quad f_0(e_1) = [9]$$

$$f_3(u_0) = [1] \quad f_0(u_1) = [2] \quad f_0(p_0) = [4] \quad f_0(e_0) = [9] \quad f_0(e_1) = [8]$$

Given that all the aforementioned morphism have a domain $\{u_0, u_1, p_0, e_0, u_1\}$ and that $\phi(t,\text{“Entity”}) = \{u_0, u_1\}$, this means that the only element of H not represented in T can be only p_0, and hence Equation 6.10 can be rewritten as follows:

$$\begin{aligned}
&\phi(\mathfrak{g},\text{“Entity”})\backslash\texttt{fold}_{[H\mapsto T]^m,\ (H_i,\alpha)\mapsto\texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto f_j\left(\phi(\mathfrak{g}_i^H,\text{“Entity”})\backslash\phi(\mathfrak{g}_i^T,\text{“Entity”})\right)\cup\beta}(\alpha)}([]) \\
&\qquad = \phi(\mathfrak{g},\text{“Entity”})\backslash\texttt{fold}_{\{f_0,f_1,f_2,f_3\},\ (f_j,\beta)\mapsto f_j\left(\phi(\mathfrak{g}_i^H,\text{“Entity”})\backslash\phi(\mathfrak{g}_i^T,\text{“Entity”})\right)\cup\beta}([]) \\
&\qquad = \phi(\mathfrak{g},\text{“Entity”})\backslash\texttt{fold}_{\{f_0,f_1,f_2,f_3\},\ (f_j,\beta)\mapsto f_j\left(\phi(\mathfrak{g}_i^H,\text{“Entity”})\backslash\phi(\mathfrak{g}_i^T,\text{“Entity”})\right)\cup\beta}([]) \\
&\qquad = \phi(\mathfrak{g},\text{“Entity”})\backslash\texttt{fold}_{\{f_0,f_1,f_2,f_3\},\ (f_j,\beta)\mapsto f_j([p_0])\cup\beta}([]) \\
&\qquad = \phi(\mathfrak{g},\text{“Entity”})\backslash\bigcup_{f_j\in\{f_0,f_1,f_2,f_3\}} f_j([p_0]) \\
&\qquad = \phi(\mathfrak{g},\text{“Entity”})\backslash[3,4] \\
&\qquad = [0,1,2,5]
\end{aligned}$$

Please note that node 5 is kept because it has not been matched by H and, consequently, it does not appear in the final morphisms. ◂

We now must add to the previously filtered vertices the new ones v'_{c+1} that are generated, for each morphism f_j, by the vertices v_c in $\phi(\mathfrak{g}_i^T)$: the id v'_{c+1} associated to the generated vertex depends on both the position p of the generalized graph grammar within the sequence of the queries, and to the id j associated to the generated morphism f_j, and the one belonging to the transformation vertex. Therefore:

$$v'_{c+1} = dt(j, v_c)_{c+1}$$

Moreover, each of the v'_{c+1} must have the same label of v_c ($\ell(v'_{c+1}) := \ell(v_c)$) and must share its same attributes ($\phi(v'_{c+1}) = \phi(v_c)$). In particular, $\phi(v'_{c+1}, k)$ must contain all the elements matched by $\phi(v_c, k)$. This implies the creation of the following object:

$$W^V_{f_j,v_c}(\eta) = \texttt{create}^{dt(j,v_c)_{c+1}}_{\ell(v_c),\xi(v_c),\bigcup_{k\in\text{dom}(\phi(v_c))}[[k,f_j(\phi(v_c,k))]]}(\eta) \tag{6.11}$$

Hereby, the final vertex set is provided as follows:

$$V' = \delta V \cup \texttt{fold}_{\mathcal{G},\ ([H_i\to T_i]^{m_i},\alpha)\mapsto\texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto\{dt(j,v)_{c+1}|v_c\in\phi(\mathrm{g}^T_i,\text{“Entity”})\backslash\phi(\mathrm{g}^H_i,\text{“Entity”})\}\cup\beta}(\alpha)}([])$$

While the creation of such newly associated elements can be performed via Equation 6.11 as follows:

$$\eta_V(\eta) = \texttt{fold}_{\mathcal{G},\ ([H_i\to T_i]^{m_i},\alpha)\mapsto\texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto W^V_{f_j,v_c}(\beta)}(\alpha)}(\eta)$$

▸ **Example 33** (continuing from p. 188). *In our case, the set of the vertices in T that are not present in H is empty, and hence no new vertices are generated in this case.* ◂

Differently to what it has been already stated for the vertices, we must preserve all the edges that are **(i)** either kept after the matching and transformation phase, **(ii)** or are connecting vertices that are either returned, or newly created, or connect vertices to other nested elements. For this reason, we must consider the newly generated edges first (since they may contain other nested vertices) prior to the definition of the edges that must be removed. Now, the newly generated vertices have to take into account that they will be associated to either existing or aggregated source and target vertices. We can distinguish them in the following expression returning their ids because the first ones are directly provided with no dovetailing definition, while the latter are defined like so:

$$\begin{aligned}\mathbf{M}_{f_j,L,i} = &\left\{\, dt(j,v)_{c+1} \;\middle|\; v_c \in L \cap \left(\phi(\mathrm{g}^T_i, \text{“Entity”})\backslash\phi(\mathrm{g}^H_i, \text{“Entity”})\right) \right\}\\ &\cup \left\{\, k \in f_j(v_c) \;\middle|\; v_c \in L \cap \phi(\mathrm{g}^H_i, \text{“Entity”}) \cap \phi(\mathrm{g}^T_i, \text{“Entity”}) \right\}\end{aligned} \tag{6.12}$$

Therefore, the aforementioned extension of W for the edges where those are associated to their new source and target vertices is defined as follows:

$$\begin{aligned} W^E_{f_j,e_c,s',t'}(\eta) &= \texttt{create}^{dtl([e_c,s',t'])_{c+1}}_{\ell(e_c),\xi(e_c),[[\text{“src”},[s']],[\text{“dst”},[t']]]\oplus\bigcup_{k\in\text{dom}(\phi(e_c))}[[k,f_j(\phi(e_c,k))]]}(\eta)\\ \widetilde{W}_{f_j,e_c,i'}(\eta) &= \texttt{fold}_{\mathbf{M}_{f_j,\phi(e_c,\text{“src”}),i'},\ (s',\alpha)\mapsto\texttt{fold}_{\mathbf{M}_{f_j,\phi(e_c,\text{“dst”}),i'},\ (t',\beta)\mapsto W^E_{f_j,e_c,f_j(s'),f_j(t')}(\beta)}(\alpha)}(\eta)\end{aligned} \tag{6.13}$$

Thus allowing to return the following new edges:

$$\eta_E(\eta_V) = \texttt{fold}_{\mathcal{G},\ ([H_i\to T_i]^{m_i},\alpha)\mapsto\texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto\texttt{fold}_{\phi(\mathrm{g}^T_i,\text{“Relationship”})\backslash\phi(\mathrm{g}^H_i,\text{“Relationship”}),(e_c,\gamma)\mapsto\widetilde{W}_{f_j,e_c,i'}(\gamma)}(\beta)}(\alpha)}(\eta_V)$$

while the edge set can be enriched as follows:

$$nE = \texttt{fold}_{\mathcal{G},\ ([H_i\to T_i]^{m_i},\alpha)\mapsto\texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto\left\{ dtl([e_c,s',t'])_{c+1} \;\middle|\; \substack{e_c\in\phi(\mathrm{g}^T_i,\text{“Relationship”})\backslash\phi(\mathrm{g}^H_i,\text{“Relationship”})\\ s'\in f_j(\phi(e_c,\text{“src”})),t'\in f_j(\phi(e_c,\text{“dst”}))} \right\}\cup\beta}(\alpha)}([]) \tag{6.14}$$

At this point, we must state that we shall keep only the edges that link vertices that are preserved, because they are either returned in δV or nested inside a nested object (either vertex or edge). In order to expand each possible nesting for each newly generated vertex (or edge) in V' (or $nE \cup E$), we use φ^* thus forcing to extract each nested component. Therefore, the set of the edges to be returned is the following one:

$$E' = nE \cup \{e_c \in \phi(\mathfrak{g}, \text{“Relationship”}) \big| \phi(e_c, \text{“src”}), \phi(e_c, \text{“dst”}) \in \\ \wp(\delta V \cup \varphi(nE \cup \phi(\mathfrak{g}, \text{“Entity”}) \cup \phi(\mathfrak{g}, \text{“Relationship”})))\} \tag{6.15}$$

▸ **Example 33** (continuing from p. 189). *All the edges in $\phi(\mathfrak{g}_i^T, \text{“Relationship”})$ are not defined in $\phi(\mathfrak{g}_i^H, \text{“Relationship”})$, and hence they are considered as new edges. Moreover, the only edge to be instantiated is ca_0 having $\lambda(ca_0) = (u_0, u_1)$. Given that both u_0 and u_1 appear in both H and T, we have that the $\boldsymbol{M}$-s in Equation 6.12 can be rewritten as follows:*

$$\boldsymbol{M}_{f_j,[u_0],i} = f_j([u_0]) \quad \boldsymbol{M}_{f_j,[u_1],i} = f_j([u_1])$$

Consequently, Equation 6.13 on the preceding page can be partially rewritten for each morphism m_j as follows:

$$\widetilde{W}_{f_j,ca_0}(\eta) = \texttt{fold}_{f_j([u_0]),\ (s',\alpha)\mapsto \texttt{fold}_{f_j([u_1]),\ (t',\beta)\mapsto W^E_{f_j,ca_0,s',t'}(\beta)}(\alpha)}(\eta)$$

and the internal expression W can be rewritten as follows:

$$W^E_{f_j,ca_0,s',t'}(\eta) = \texttt{create}^{dtl([e,s',t'])_{c+1}}_{[\texttt{coauthorship}],[],[[\text{“src”},[s']],[\text{“dst”},[t']],[\text{“pp”},[p_0]]]}(\eta)$$

In order to provide further simplifications, we must introduce all the morphisms. Therefore, Equation 6.14 on the previous page can be rewritten as follows:

$$\begin{aligned}
nE &= \texttt{fold}_{\mathcal{G},\ ([H_i\to T_i]^{m_i},\alpha)\mapsto \texttt{fold}_{m^i_{H_i}(\eta),\ (f_j,\beta)\mapsto \left\{ dtl([e,s',t'])_{c+1} \,\middle|\, \substack{e_c\in\phi(\mathfrak{g}_i^T,\text{“Relationship”})\backslash\phi(\mathfrak{g}_i^H,\text{“Relationship”}) \\ s'\in f_j(\phi(e_c,\text{“src”})),t'\in f_j(\phi(e_c,\text{“dst”}))} \right\}\cup\beta}(\alpha)}([]) \\
&= \texttt{fold}_{\{f_0,\dots,f_3\},\ (f_j,\beta)\mapsto \left\{ dtl([e,s',t'])_{c+1} \,\middle|\, \substack{e_c\in\phi(\mathfrak{g}_i^T,\text{“Relationship”})\backslash\phi(\mathfrak{g}_i^H,\text{“Relationship”}) \\ s'\in f_j(\phi(e_c,\text{“src”})),t'\in f_j(\phi(e_c,\text{“dst”}))} \right\}\cup\beta}([]) \\
&= \bigcup_{f_j\in\{f_0,\dots,f_3\}} \left\{ dtl([e,s',t'])_{c+1} \,\middle|\, \begin{matrix} e_c \in \phi(\mathfrak{g}_i^T, \text{“Relationship”})\backslash\phi(\mathfrak{g}_i^H, \text{“Relationship”}) \\ s' \in f_j(\phi(e_c, \text{“src”})), t' \in f_j(\phi(e_c, \text{“dst”})) \end{matrix} \right\} \\
&= \bigcup_{f_j\in\{f_0,\dots,f_3\}} \left\{ dtl([e,s',t'])_{c+1} \,\middle|\, s' \in f_j(\phi(ca_0, \text{“src”})), t' \in f_j(\phi(ca_0, \text{“dst”})) \right\} \\
&= \bigcup_{f_j\in\{f_0,\dots,f_3\}} \left\{ dtl([e,s',t'])_{c+1} \,\middle|\, s' \in f_j([u_0]), t' \in f_j([u_1]) \right\} \\
&= \{ dtl([ca_0,0,2])_1, dtl([ca_0,2,0])_1, dtl([ca_0,2,1])_1, dtl([ca_0,1,2])_1 \}
\end{aligned}$$

Consequently, we can see that four edges are returned, one for each morphism. We now must consider the edges that we want to keep, that are the edges that still link vertices that are preserved by the transformation or that are nested: given that $\delta V = [0,1,2,5]$, that the newly generated vertices do not nest any content and that the only elements that nest some elements are the edges in nE containing the matched papers having id 3 and 4, we have that the edges $E'\backslash nE$ that have been preserved are all the edges appearing in the former nested graph, and hence:

$$[6,7,8,9,10]$$

As this example finally showed, this allows to nest the matched elements inside either vertices the edges (as in this example) only if explicitly stated by the graph transformation. ◂

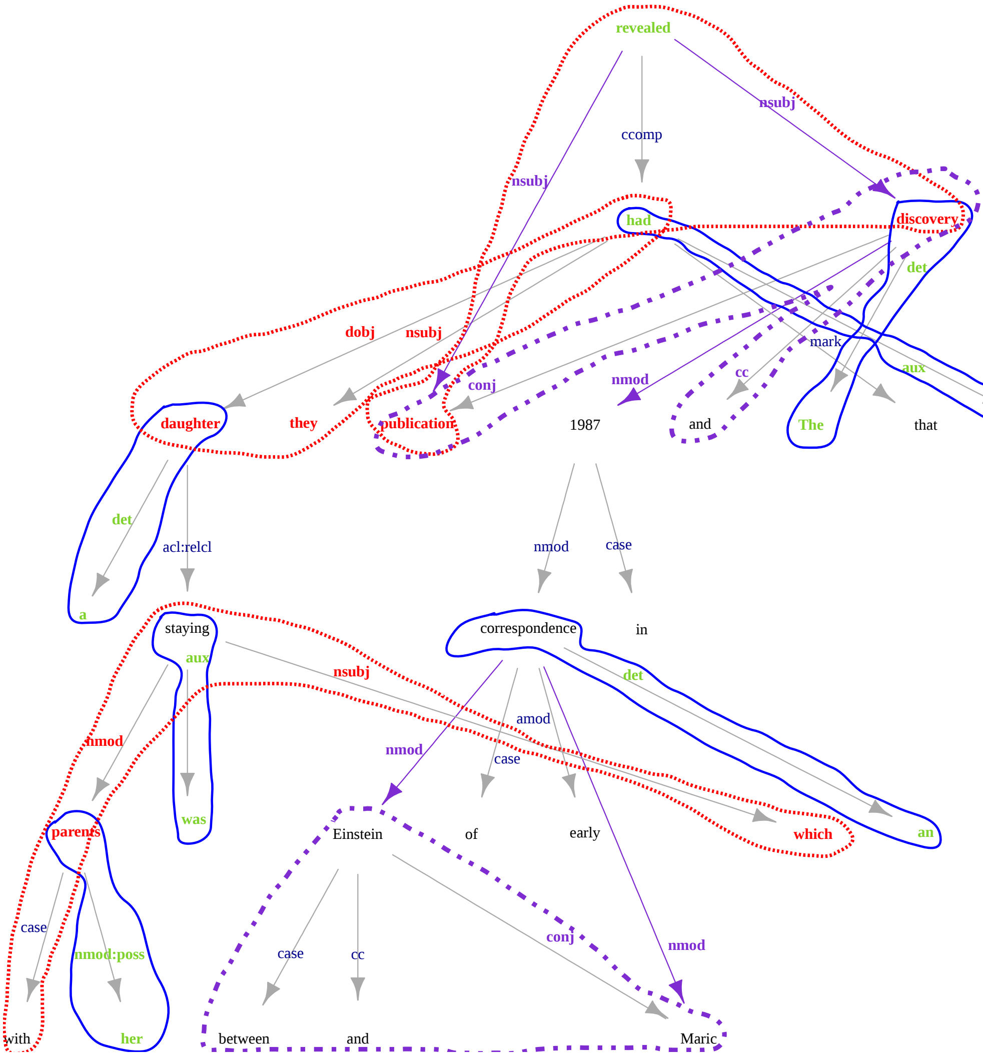
We can now continue with the formal definition of the generalized graph grammar:

▸ **Definition 66** (continuing from p. 184). *Given a nested graph $\eta = (\mathfrak{g}, O, \ell, \xi, \phi)$, the application of the set of rules $\mathcal{G}$ on η is defined as follows:*

$$\mathcal{G}(\eta) = \texttt{elect}_{\omega}(\texttt{create}^{\omega\leftarrow\max O+1}_{[],[],[[\text{"Entity"},V'],[\text{"Relationship"},E']]}(\eta_E(\eta_V(\eta))))$$

◂

At this point, we can use this same operator to perform more than one changes at once: for example, we can simply extend the previous example in order to associate to each user the number of the papers that he has coauthored by simply using such expression either in a containment or in a ξ expression.



■ **Figure 6.6** Underlying which part of the dependency graph are matched by the patterns outlined in Figure 6.5a on page 186. The blue matches represent the first graph matching rule, the purple ones with the loose dashing represent the second one, while the last one with the thick dashes represent the last one.

▸ **Example 34.** *Despite the possibility of performing multiple matches and rewriting as the previous examples showed, this approach is not always possible. After explicitely defining what a generalized graph grammar is, let us suppose to want to match simultaneously all the pattern graphs that were previously provided. The result of such matching phase is then depicted in Figure 6.6: if we now compare the matches with the expected transformations, we can see that some times we would return both matched contents and aggregated ones while, on the other hand, we would like that the changes provided by one of the grammar rules are kept in the following matching phases. You can test such outcomes by using the* `Sherlock` *project.*

In order to avoid such conflicts, the only thing that is possible to do is to subdivide one single generalized graph grammar rule into more subsequent matching and rewriting steps. The next chapter will provide an example of such operator, that will allow us to generalize the nesting for semistructured and nested relational data for graphs.

The nesting operator for nested relations [LY05, PVG92] allows nesting other attributes belonging to a same relation inside a field belonging to the same relation. If we look at the definition of a nested graph, vertices and edges are not explicitly defined as values associated to an attribute of a specific object inside database. Consequently, the traditional nesting operation for nested relations cannot help us for the definition of an operator allowing to store a subgraph of a given nested graph inside either a vertex, or another edge. We must also note that we cannot create new vertices matched by different graphs H_i and link them by newly created edges by using a link discovery predicate over such predicates, we must necessarily apply a second operator (either a link discovery or a generalized graph grammar operator) for the creation of such edges.

Therefore, even this operator cannot be used in practice to define a general operator allowing graph nesting. The next chapter is going to provide another possible solution for graph nesting by incorporating the definition of the semistructured grouping operators and by allowing vertices' and edges' grouping contemporarily.

Before closing this chapter, we should ask ourselves whether this operator actually matches within the requirements of ϙ, that already appeared in Section 2.3.3 on page 50 and 5.4.2 on page 143, where we also provided an algorithm for transforming the morphisms associated to the source schema into morphisms associated to the hub schema. As we can see, this approach is more general than the previous one, because through $\mathcal{G}$ we can even create new elements as aggregations of the matched elements which have not been returned by the graph transformation phase. Notwithstanding, this operator provides more data than the one required by the data integration scenario: the non matched objects within the head pattern are returned, while in the previous chapter's approach we only return the matched objects, and we distinguish different possible matchings between the head and tail pattern. First, if we want to remove the unmatched elements from the final vertex and edge set, we can simply restate the former definition as follows:

$$\text{ϙ}^{\mathcal{G}}_{\mathcal{H},\mathcal{T}}(\mathcal{H})(\eta) = \texttt{elect}_{o_{c+1}}(\texttt{create}^{\textbf{let}\ o_c=\max O\ \textbf{in}\ o_{c+1}}_{[],[],[[\text{“Entity”},\delta V],[\text{“Relationship”},nE]]}(\eta_E(\eta_V(\eta))))$$

Moreover, if we have simple GSMs not represented as nested graphs, we can simply ignore the returned edges. Now, this last equation differs from the previous proposal from qualitative measures, but they have no specific distinctions from a formal point of view (that is, they both adhere to the specifics, but they substantially differ on the returned result). Therefore, further tests have to be carried out in order to test the quality of both ϙ definitions, in order to check which of the two provide the best expected results from a user experience point of view. We address to some future work on establishing which of the two definitions is better for data integration within nested data.

6.4 Conclusons

This chapter provided the definition of the GSQL query language expressed through an algebra, which was able to express set, relational, semistructured and (nested) graph operators. In particular, it was showed that such operators were able to implement all the missing operators required for the GAV data integration scenario, which are the schema extraction (via aggregation) and the matching with rewriting (ϙ) by generalizing the properties of graph rewriting and pattern matching. On the other hand, we do not still provide a full definition of all the possible graph data operators but we showed that it is possible to combine the semistructured operations with the edge creation ones to create new nested graphs.

We also remark some futher works that should be carried out on GSQL:

- Provide some equivalence rules for GSQL if any.
- If there are some equivalence rules, provide some query rewriting optimizations as it currently happens on relational models' query plans.
- Similarly to the relational $\bowtie_\theta$, check if there are other set of operators that can grouped and provide better optimizations within one single definition that in multiple disaggregated operators.
- Provide a more formal definition for a nested graph traversal language, combining XPath's and NautiLOD's semantics.
- As it will be outlined in the next chapter, check if there are some cases when traversals may be provided as a predicate for a GSQL operator and, given that both graph traversal and GSQL are represented in GSQL itself, check if the former types of optimization provide some benefit.

Last, we also showed how such algebra allows the definition of graph joins. In the next chapter, we're going to introduce an algorithm for a specific instance of the graph nesting operator. By showing that both filtering queries and algebraic operators may be expressed within the same language, it would be also possible to perform mutual optimizations between such operators: this intuition will lead into the next chapter, where it will be showed how graph pattern matching queries can be nested within graph nesting operators. This final scenario will motivate the need of such a low level algebra for describing each possible optimizable step, thus leading towards some more future works on higher level operation over GSMs. This possibility is going to concretely lead to better optimizations and, therefore, we're going to create an algorithm, where both graph visit and new graph creations are combined together.

7 On Nesting Graphs

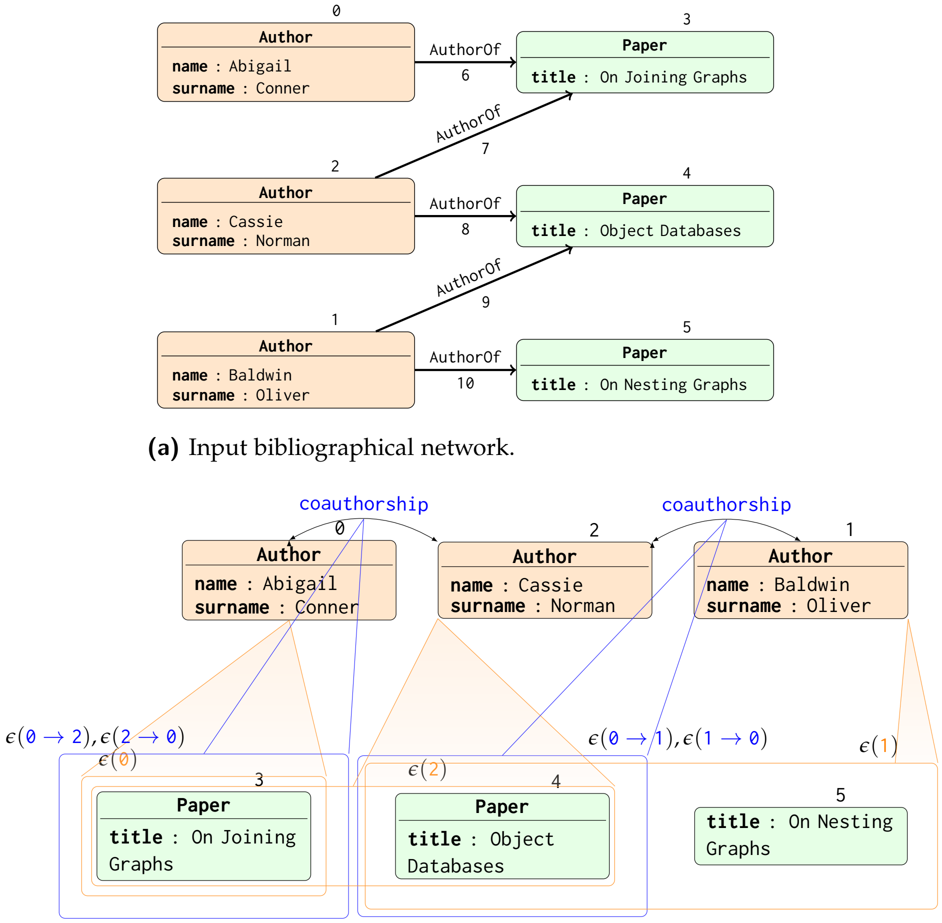
Contents

> *Did she say: “He yelled, ‘Frame stories are an example of nestings in literature!’ ”?.*
>
> — An example of a nested quote.

Graphs allow flexible analyses of relationships among data objects. Thus, graph data management systems play an increasing role in present data analytics. Graphs have been already used as a fundamental data structure to represent data within different contexts such as corporate data [RHKB13, PAK16], social networks [XKS13, BK14] and linked data [VTBL13]. Despite an increasing number of applications, a general operator that aggregates a single graph in a roll-up fashion is still missing. The operation of adding structural aggregations to an existing graph is called *graph nesting*. A respective operator shall not only create a new graph of *nested vertices* and *nested edges*, each containing subgraphs of the original input graph, but also preserve the vertices and edges that are not affected by the actual operation. Further on, the operator must ensure that the nested elements can be freely unnested such that the original graph may be obtained back again. Vertices or edges of the original graph will be called *members* of a nested vertex or edge, if they appear in its underlying subgraph.

▸ **Example 35.** *Figure 7.1a represents a bibliographic graph with (at least)* Authors *and* Papers *as vertices and* authorOf *relationships as edges, which connect authors to papers they have authored. With the graph nesting operator, we want to "roll up" the graph into a coauthorship graph (Figure 7.1b): each* Author *will be connected by a* coAuthor *edge with another* Author(2) *if they have published at least one paper in common. More precisely, each resulting* Author(2) *vertex shall contain authored papers as vertices and each* coAuthor *edge shall contain all the coauthored papers with regard to source and target* Authors. *However, we want to exclude* coAuthor *hooks over the same vertex.*

In a resulting nested graph, edges connecting nested vertices express that members of the nested vertices are connected by an edge or, more generally, by a path in the original graph. In contrast to this general approach, current literature distinguishes between *vertex summarization* and *path summarization*. Thus, it is not possible to define a single query evaluating both kinds of patterns at the same time. Before outlining our proposed algorithmic solution, let's have a look on these existing approaches:

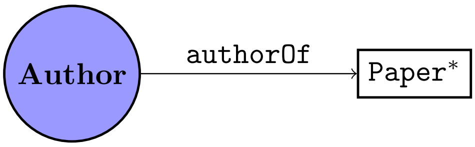


(a) Input bibliographical network.

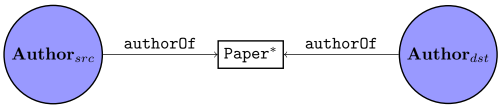
(b) Nested result: given two Authors a and a', there exist two coauthorship edges, $a \to a'$ and $a' \to a$ if and only if they share some authored paper contained respectively in $\phi(a \to a', \text{“Entity”})$ and $\phi(a' \to a, \text{“Entity”})$. Moreover, each author a is associated to the set of his authored papers $\phi(a, \text{“Entity”})$.

■ **Figure 7.1** Nesting a bibliographic network. While in the previous Example 7 on page 41 the information was summarized, in this case the provenance information is nested within the original node.

The *vertex summarization* strategies group vertices in the manner of the relational GROUP BY operation and aggregate edges accordingly [JPR17]. In this class of operations, summarized edges can only be formed by edges that directly connect members of summarized vertices in the original graph. In other words, these approaches cannot freely nest edges: for example, it is not possible to aggregate paths. Since most of vertex summarization techniques are based on graph partitioning, they further provide no support for nested vertices and edges with overlapping members [YG16, THP08, JFL15]. Exceptions are HEIDS [CJQ16] and Graph Cube [ZLXH11], which perform graph summarizations of one single graph over a collection of non pairwise disjoint subgraphs. However, the union of these underlying subgraphs must be equivalent to the original graph, i.e., it is not possible to take vertices and edges of the original graph over to the summarized graph or to represent outliers that belong to no group.



(a) Vertex summarization pattern (V). Author is the vertex grouping reference γ_V.



(b) Path summarization pattern (E). Author$_{src}$ and Author$_{dst}$ are respectively edge grouping references γ_E^{src} and γ_E^{dst}.

■ **Figure 7.2** Vertex and Path summarization patterns for the query expressed in the running example. Vertex and edge grouping references are marked by a light blue circled node. As we can see, the vertex grouping reference depicts the same property expressed by edge grouping references.

By contrast, *path summarization* techniques allow the aggregation of multiple paths among pairs of source and target vertices that share the same properties. Currently, approaches to path summarization can only be found within graph query languages. These languages also support vertex summarization, but no combination of both approaches in a single step. Cypher, the query language of the productive graph database Neo4j, can perform distinct aggregations only within distinct `MATCH` clauses. SPARQL, the standard query language of the resource description framework (RDF), requires to combine vertex and path aggregation with a `UNION` operator, i.e., the same input graph must be visited twice.

In particular, vertex nesting approaches were not compared to already existing (graph) query languages and, consequently, we provide a general graph nesting definition in Section 60 on page 175 and its generic implementation in Section 7.1.2 on page 201 as the first naïve (but general) algorithm for a graph nesting. Such straightforward implementation proves to be inefficient: if we want to nest k subgraphs of g within g itself, in the worst case scenario we have a visiting cost of $O(|g|^k)$. This results in an exponential algorithm, because the size of k may vary, while $|g|$ is fixed. This implies that the graph must be always visited more than once, even if this may not be required. This general operator also proves to be inefficient in practice, it allows detecting a broader class of problems and optimizable algorithms. In order to reduce the graph visiting cost from $|g|^k$ to $O(|g|)$, we could use a graph traversal approach: instead of pre-computing k subgraphs of g that are going to be later on used to nest g, we can directly perform the graph nesting while visiting the graph, thus allowing not to perform additional costs for comparing the resulting graphs in a later step. The following example shows how such queries can be efficiently formulated and implemented.

▸ **Example 35** (continuing from p. 195). *Figure 7.2 shows summarization patterns to describe the vertex (V) and path (E) nestings of our bibliographic network example: the former will create a nested* Author(2) *vertex and the latter will create a* coAuthor *nested edge. Given that V appears twice in E, we may also pre-istantiate the pattern V by visiting E once. The two patterns have different key roles: while the vertex summarization retrieves all the* Papers *that one* Author *has published and nest them within one single matched* Author, *the path summarization nests all the*

*Papers authored by two different Authors as members of a newly created nested edge connecting the two previously nested vertices. We can express this nesting requirement within vertex and path summarization patterns by electing both Author in V as a **vertex grouping reference** γ_V, and the two distinct Authors in E (acting as the nested edge's source and targets) respectively as **edge grouping references** γ_E^{src} and γ_E^{dst}. In particular, the two latter vertices must both match with the vertex grouping references, so that the newly generated edge will have as source and target the previously vertex-nested elements. In particular, this chapter focuses on E where γ_E^{src} and γ_E^{dst} are separated by a two-edge (hop) distance.*

We solve this problem by visiting the graph only once: If the current vertex is a Paper, traverse backwards all the authorOf edges, thus reaching all of its Authors (γ_E^{src} and γ_E^{dst}), that are going to be coAuthors for at least the current chapter. Instead of associating the nesting content at the end of the graph visiting process, I can incrementally define the subgraph to be nested by using a separated nesting index: by visiting the two distinct Author vertices adjacent to the current Paper, the latter one shall be contained in both final Author(2) vertices, thus allowing the definition of a coAuthor edge. By doing this, only the edges are visited twice, but the vertices are visited only once. These patterns allow to reach the optimal solution.

There might be other possible patterns that can be optimized, but we're going to focus on vertex and path summarization patterns where edge grouping references are connected to each other at a 2 edge step distance (Section 7.4.1). We're also going to show how such optimizations can be detected beforehand by looking at the pattern representation. This chapter shows that such query language limitations can be reduced by using a graph nesting operator, which performs both vertex and path summarization queries concurrently with only a single visit of the input graph. We propose graph patterns to declare graph nesting operations and propose algorithmic optimizations as well as a specific physical model for efficient execution.

In the remainder of this paper, we will This is achieved by the following contributions:

- We propose a **Nested Graph Data Model** that is capable to implement the aforementioned solution of our example scenario. We use an optimized physical model that differs from the logical one (Section 7.3).
- We provide a general definition of a **Graph Nesting Operator** which combines vertex and path summarization approaches to nesting graphs (Section 7.4).
- We introduce the **Two HOp Separated Patterns (THoSP)** algorithm for graph nesting (Section 7.4.1). We present the results of an experimental evaluation that compare it to alternative implementations using graph (SPARQL, Cypher), relational (SQL) and document oriented (AQL) query languages: our solution outperforms all competitors by at least one order of magnitude in average with regard to the sum of both indexing and query evaluation time (Section 7.5).
- A general strategy on how to extend the THoSP algorithm for patterns having vertex and edge grouping references is provided (Section 7.2). This approach shows that graph nesting can be defined on top of current-existing graphs extraction languages.

The repository at `http://rebrand.ly/unibo612df` provides the C++ benchmarks' source code.

7.1 Graph Query Languages limitations' on Graph Nesting

7.1.1 Graph Joins' limitations in providing the $\nu_{\cong}$ operator

We now discuss a use case where the graph nesting approach aggregates similar nodes into one single representation while discarding the original sources' pieces of information. Please refer to Section 2.3.3 on page 50 for the general data integration scenario where such operator ($\nu_{\cong}$, that is grouping over an equivalence classifier $\cong$) may be adopted. In particular, we are going to show that, even though full graph join may represent this procedure, the resulting solution may be hardly implementable. To make our point, we are going to provide a different use case from the one previously offered.

▸ **Example 36.** *Suppose to integrate, within a graph ETL, three distinct bibliographic sources (e.g. DBLP, Microsoft Academic Graph, Google Scholar) into one final graph. Each of these separately undergoes a data cleaning phase and are represented as the distinct connected components after an entity resolution processes [NHNR17]. Such components are represented by the graphs G_1, G_2 and G_3 in Figure 7.3a. As a next step, the entity resolution [SPR17] analyses if nodes are appearing in the different sources and representing the same entity: as a consequence, we create the red relations in Figure 7.3a, and then collect them into graphs representing a clique of all the resolved entities. The outcome of this operation is provided in Figure 7.3b on the next page, where all the entities representing one single entity are merged into one single vertex.*

Another example where such edge joins could be of some use is within the ontology alignment process provided in Definition 5 on page 49. For instance, we can interpret each description logic axiom $C \sqsubseteq C'$ as an oriented edge connecting each vertex of the graph satisfying the predicate C to the ones satisfying C', thus allowing to join two graphs which schemas were previously aligned.

The class of graph $\otimes_\theta$ products could be then generalised to support edges E as a basis for the definition of the θ predicates. In particular, such predicate θ_E can be defined over a set of edges E as follows:

$$\theta_E(a,b) \Leftrightarrow \exists e \in E.\lambda(e) = (a,b)$$

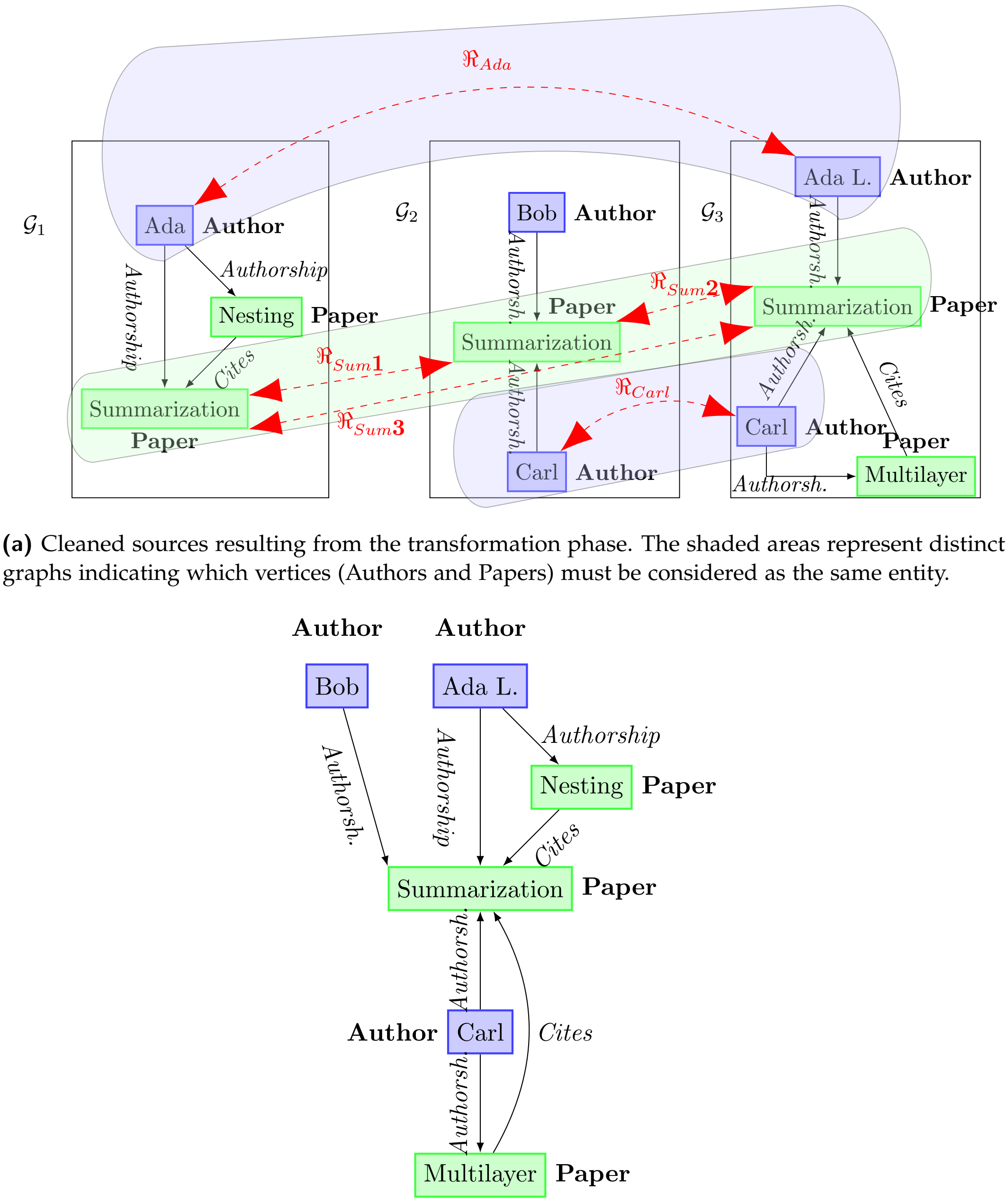
In such cases, we will write the predicate θ_E directly as E through abuse of notation allowing to list the edges involved within the join operation directly. We can now ask ourselves if the following expression provides G_{out} in Figure 7.3b:

$$(G_1 ⟗^{\vee}_{\{\mathfrak{R}_{Sum1}\}} G_2) ⟗^{\vee}_{\{\mathfrak{R}_{Ada},\mathfrak{R}_{Sum2},\mathfrak{R}_{Carl}\}} G_3$$

Let us first perform $(G_1 ⟗^{\vee}_{\{\mathfrak{R}_{Sum1}\}} G_2)$ as G_{12}, thus merging the two `Summarization` nodes in G_1 and G_2 into a node s_J: we can see that we can still join the nodes linked by the edges $\mathfrak{R}_{Ada}$ and $\mathfrak{R}_{Carl}$, but we can no more join s_J with the `Summarization` vertex in G_3, because $\mathfrak{R}_{Sum2}$ is not defined on s_J. As a consequence, we have that this way to join graphs is no more associative: if we now associate the join to the right and evaluate the following expression:

$$G_1 ⟗^{\vee}_{\{\mathfrak{R}_{Sum1}\}} (G_2 ⟗^{\vee}_{\{\mathfrak{R}_{Ada},\mathfrak{R}_{Sum2},\mathfrak{R}_{Carl}\}} G_3)$$

we can now see that $\mathfrak{R}_{Sum1}$ is not defined over the `Summarization` node from G_1 and the merged node from $G_2 ⟗^{\vee}_{\{\mathfrak{R}_{Ada},\mathfrak{R}_{Sum2},\mathfrak{R}_{Carl}\}} G_3$. As a consequence, these two evaluations of the edge join query provide two different results, which is disadvantageous within an



(a) Cleaned sources resulting from the transformation phase. The shaded areas represent distinct graphs indicating which vertices (Authors and Papers) must be considered as the same entity.

(b) Representing the expected output G_{out} for the data integration phase, ready to be fit to the graph datawarehouse.

■ **Figure 7.3** $\nu_{\cong}$: another use case requiring the nesting operator ν, as required by the last step of the data integration process before actually performing the queries over the integrated data. This last chapter ends the thesis by providing an example of the last operator required for such data integration process. In this case the outcome of the clustering operator is a similarity predicate $\cong$ using a entity-resolution process.

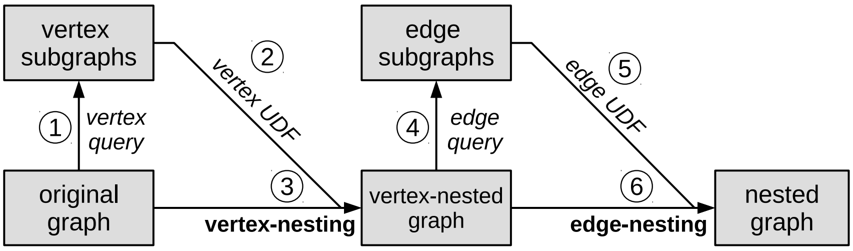


Figure 7.4 Overview about the general graph nesting process. UDF stands for User Defined Function.

automated computer environment requiring operations that can be easily scaled by using associativity rules (i.e., order invariant) and that can be hence parallelised.

Figure 7.3a also suggests on how such an operation can be carried out: instead of performing the stepwise full outer join on the graph, we can first set-union the three graphs, and then aggregate all the elements within the same dashed area as one single vertex by using the ν nesting operator over graphs. This approach motivates the need of such graph nesting operator.

7.1.2 Implementing Graph Nesting over (two) graph collections

In order to create nested vertices and edges, we can first propose a generic *nesting operator*, allowing the creation of arbitrary vertices and edges over a given (nested) graph (**original** or **input graph**). Figure 7.4 provides an overview about the nesting process. First, in step ① a *query* is used to determine subgraphs which will later on be nested as vertices' content. In Example 35 on page 195, we want to nest each Author with its egonet represented by the paper he has authored, and hence we're going to extract all the subgraphs matching the pattern Author $\xrightarrow{\texttt{authorOf}}$ Paper* (Figure 7.2a). Another example is the Social Network scenario presented in Example 6 on page 41: we want to nest each user of the `DataSource` in her or his `Community`. Here, the term *query* generally refers to either a pattern matching query (e.g., a community can defined by each user's egonet) or a partitioning algorithm (e.g., community detection). As a consequence, the way to extract subgraphs to be summarized as vertices is application dependent, and may require a previous graph traversal phase or not.

Subsequently, in step ② a User Defined Function (UDF) can be applied to the resulting subgraphs in order to derive aggregated vertices from the subgraph's elements that will be added to the final nested vertex. In particular, in a data integration scenario one could select representative property values to appear in the resulting vertex or, in a summarization process, one could evaluate measures such as vertex count over the extracted graph. The subsequent step ③ takes the original graph as well as the processed subgraphs as input and turns the latter into nested vertices. The resulting vertex-nested graph contains all the nested vertices from ③, containing at least one vertex also nested in the original graph, but also all vertices of the original graph not appearing in one of the subgraphs. In the Social Network scenario, the previous steps could have been performed by an external community detection algorithm.

In order to obtain the final nested graph from the vertex-nested graph, we must apply a similar process over the edges: in step ④ subgraphs containing nested vertices can be extracted using a dedicated query. These subgraphs form the basis of the later nested

Algorithm II.7 General Nesting Algorithm

```
procedure GeneralNesting(($\mathcal{G}_V$, UDF$_V$), ($\mathcal{G}_E$, UDF$_E$), keep; ɳ)
    $GV = \varnothing;\quad GE = \varnothing;\quad VR = \varnothing;\quad ER = \varnothing$
    for each graph $g_v = (V, E)$ in $\mathcal{G}_V$ do
        $\delta V = V \cap \phi(g, \text{“Entity”})$
        $\delta E = E \cap \phi(g, \text{“Relationship”})$
        if $\delta V \neq \varnothing \vee \delta E \neq \varnothing$ then
            $GV = GV \cup \{UDF_V(g_v)\}$
            if keep then
                $VR = VR \cup \phi(g, \text{“Entity”}) \backslash \delta V$
                $ER = ER \cup \phi(g, \text{“Relationship”}) \backslash \delta E$
    for each graph $g_e = (V, E)$ in $\mathcal{G}_E$ do
        $\delta V = V \cap \phi(g, \text{“Entity”})$
        $\delta E = E \cap \phi(g, \text{“Relationship”})$
        if $\delta V \neq \varnothing \vee \delta E \neq \varnothing$ then
            $\varepsilon = UDF_E(g_e)$
            if $\lambda(\varepsilon) \in (GV \cup \phi(g, \text{“Entity”}))^2$ then
                $GE = GE \cup \{\varepsilon\}$
                if keep then
                    $VR = VR \cup \phi(g, \text{“Entity”}) \backslash \delta V$
                    $ER = ER \cup \phi(g, \text{“Relationship”}) \backslash \delta E$
    return $(GV \cup VR, GE \cup ER)$
```

edges where, using our running example, they can be previously extracted by visiting the pattern Author $\xrightarrow{\texttt{authorOf}}$ Paper* $\xleftarrow{\texttt{authorOf}}$ Author (Figure 7.2b). In contrast to vertices, an UDF (step ⑤) is mandatory since its purpose is not only to aggregate properties but also to select source and target on the later nested edge of step ⑥. These can be both nested and simple vertices. For example, in a summarization scenario a nested edge may connect two previously nested vertices while in a transformation scenario it can replace a path between two simple vertices. Consequently, before adding edges between vertices that do not appear in the vertex-nested graph, we must also ensure that such UDF generate valid edges. Please note that, also in this case, phases ④ and ⑤ can be also carried out by an external algorithm.

The unfeasibility of this naïve approach is remarked by Algorithm II.7: given that any to-be-nested graph may come from an external source, it is required that each graph collection pertaining either to the to-be-nested vertices (①-③) or edges (④-⑥) must be analysed in order to get which elements are matched within the vertices and edges of the nested graph ɳ. Only after this second visiting process, we can then apply the UDF function over the filtered graphs. This means that we must at least visit ɳ after each collection, with a cost of $|ɳ|^{|\mathcal{G}_V|} + |ɳ|^{|\mathcal{G}_E|}$, where $|\mathcal{G}_V|$ and $|\mathcal{G}_E|$ are the graph collections generated respectively from the vertex and the edge subgraph extraction process. The expensiveness of such approach is going to be later on compared to a more efficient algorithm for a specific subproblem (THoSP, Section 7.4.1). This cost must be further on increased if we want to preserve the un-matched vertices and edges in the final graph (**keep**=**true**). Therefore, we want to decrease this computational post whenever possible: given that nesting is a specific case of summarization[1], we can observe that the previously proposed graph summarization algorithms provide one possible algorithmic enhancement for the cases when the graph is represented by an exact partition, as when we group the graph by either its labels or attributes [JPR17]. As previously observed, the purpose of this chapter is instead to extend

[1] Compare this statement with the formal definitions of semistructured and relational nesting proposed in Definition 52 on page 167.

the family of such graph nesting algorithms by combining pattern matching approaches to the nesting process: instead of generating multiple graph collections with an external method for then nesting them, we want to compactly generate them while visiting the graph. Then, the visited graph will be immediately used to generate a nesting, thus combinating the graph visit process with the nesting one, thus allowing to decrease the overall time complexity. We're going to tackle this problem in Section 7.2 on page 206 after providing a formal definition of such operator in Section 7.4 on page 211.

7.1.3 Query Languages' and data models' limitations

As previously mentioned, some recent query languages support graph nesting semantics by exploiting the underlying data model's features. To express the query presented in our running example, these query languages must support id collections or nested representations. For these reasons, we firstly select PostgreSQL which, by extending the SQL-3 syntax and by allowing JSON data, provides an `array_agg` aggregation function. The latter collects (i.e., groups) the result-set into arrays. We represent graphs by storing edge triples as `Edge`(*edgeId*, *sourceId*, *edgeLabel*, *targetId*). In our running example, nested vertices are obtained by grouping edges by *sourceId* and collecting all target's id results via `array_agg`. Similarly, our nested edges are obtained by joining consecutive `Edges` and then grouping by distinct *sourceId* and *targetId* vertices limiting the two hop path; the list of all the PAPERS is collected via `array_agg`. The overall graph nesting cannot be created in one single SQL query, because we cannot distinctively group the same dataset in different ways. Instead, we must perform two distinct aggregations (see Listing 7.1). We want now to discuss how current query languages can express nesting constraints within their data model of choice. In particular, we must select query languages that either support collections or nested representations allowing to express the same query presented in our running example.

All the other query languages are going to be affected by the same problem, SPARQL included (Listing 7.2): despite the fact that SPARQL may represent the graph nesting query as a single statement, a `UNION` clause implies a separate visit for the two graph patterns. The first pattern presented in Listing 7.2 (see Appendix) allows to traverse those patterns matching the coauthorship statement in Figure 7.2b, so that they can be nested within the created COAUTHORSHIP edge. The second part returns the remaining PAPER associations that have been authored by one single AUTHOR. In the first case, the edge nesting is performed via the association of different `<http://contains.io/nesting>` properties departing from one single COAUTHORSHIP edge (`?newedge`). In particular, the `OPTIONAL. . . FILTER(!bound(. . . ))` syntax is adopted instead of `FILTER NOT EXISTS`, because the latter is only supported in SPARQL1.1, which is not supported by the current version of *librdf* used to query Virtuoso.

We extend our query languages comparison presented on graph joins (Chapter 4) with AQL, because ArangoDB is a document-oriented NoSQL database, which query language AQL allows the access and the creation of nested members. An example of how such graph nesting query can be carried out in AQL is presented in Listing 7.3: in this scenario we assume that we've previously loaded our graph data with the default ArangoDB format, where vertices are indexed by id while edges are also indexed by source and target vertex id[2]. Even though AQL returns JSON documents instead of relational tables, we can state

[2] https://docs.arangodb.com/3.2/Manual/Graphs/

Listing 7.1 Graph Nesting in PostgreSQL. Two distinct tables are created for both vertices and edges. The nesting is represented by nesting the elements' id within an array (array_agg). Please note that such sybtax is not SQL-3 standard.

```
-- Nesting Vertices
SELECT distinct T.sourceId as src, array_agg(distinct T.dst) as
   ↪ papers
FROM edges-i as T
GROUP BY T.sourceId;

-- Nesting Edges
SELECT distinct T.sourceId as src, T1.sourceId as dst,
       array_agg(distinct T1.targetId) as papers
FROM edges-i as T, edges-i as T1
WHERE T.targetId = T1.targetId AND T.sourceId <> T1.sourceId
GROUP BY T.sourceId, T1.sourceId;
```

Listing 7.2 Graph Nesting in SPARQL. Given that the RDF List solution is inefficient and that named graphs cannot be used either in the RDF models as vertices or edges, we use other properties to associate to either vertices and edges the nesting content.

```
CONSTRUCT {
  ?autha ?newedge ?authb.
  ?newedge <http://contains.io/nesting> ?paper1.
  ?authc <http://test.graph/graph/edge> ?paper2.
} WHERE {
  {
    GRAPH <http://test.graph/graph/i/> {
      ?autha <http://test.graph/graph/edge> ?paper1.
      ?authb <http://test.graph/graph/edge> ?paper1.
    }
    FILTER(?autha != ?authb).
    BIND(URI(CONCAT("http://test.graph/graph/newedge/",STRAFTER(
       ↪ STR(?autha),"http://test.graph/graph/id/"),"-",STRAFTER
       ↪ (STR(?authb),"http://test.graph/graph/id/"))) AS  ?
       ↪ newedge).
  } UNION {
    GRAPH <http://test.graph/graph/i/> {
      ?authc <http://test.graph/graph/edge> ?paper2.
    }
    OPTIONAL {
      ?authd <http://test.graph/graph/edge> ?paper2.
      FILTER (?authd != ?authc)
    }
    FILTER(!bound(?authd))
  }
}
```

Listing 7.3 Graph Nesting in ArangoDB using AQL as a query lanugage. Please note that all the fields marked with an underscore represent externally indexed structures and. Therefore, only external indices are used within the query plan.

```
-- Nesting vertices
FOR b IN authorOf
  COLLECT au = b._from INTO groups = [ b._to ]
  RETURN {"author" : au,  "papers": groups }

-- Nesting edges
FOR x IN authorOf
  FOR y IN authorOf
    FILTER x._to == y._to && x._from != y._from
    COLLECT src = x._from, dst = y._from INTO groups = [ x._to ]
    RETURN {"src": src, "dst": dst, "contain": groups}
```

Listing 7.4 Graph Nesting in Neo4J using Cypher as a Query Language. Please note that, even in this case, is it not possible to return one single nested graph immediately, and hence the nested vertices must be created before creating the nested edges. This implies that a greater number of joins is required to associate the previously nested data to the original operand.

```
MATCH (a1: Author)-->(p1:Paper)
WITH a1, collect(p1.UID) AS papers1
CREATE  p=(:Authors {authored: papers1, id:a1.UID})

MATCH (a1: Author)-->(p:Paper)<--(a2: Author), (a1p: Authors), (
   ↪ a2p: Authors)
WITH a1, a2, a1p, a2p, collect(p.UID) AS common
WHERE a1.UID = a1p.id AND a2.UID = a2p.id AND a1.UID <> a2.UID
CREATE p=(a1p)-[:Papers {coauthored: common}]->(a2p)
```

that its resulting query plan is similar to PostgreSQL's query plan, except that JSON documents are returned instead of relational tables containing JSON arrays.

Last, Listing 7.4 provides an example of Cypher: even if Neo4J's property graph model does not directly nest graphs inside vertices or edges, we can associate *member* ids to each of them. This solution can be achieved by first matching the vertex summarization pattern of Figure 7.2a and then performing an Author group by (`with`). Afterwards, we nest the set of authored Papers via `collect`. Last, we match the path summarization pattern presented in Figure 7.2b and group it by source and destination Author, then we create the coAuthorship edge containing the co-authored Paper's id. As it will be observed within the benchmarks (see Section 7.5), the solution of not separating the elements' ids from their data quickly leads to an intractable solution.

In all the former query languages, the vertices undergoing `GROUP BY`-s are either vertex or edge grouping references.

7.2 Class of optimizable graph nesting queries

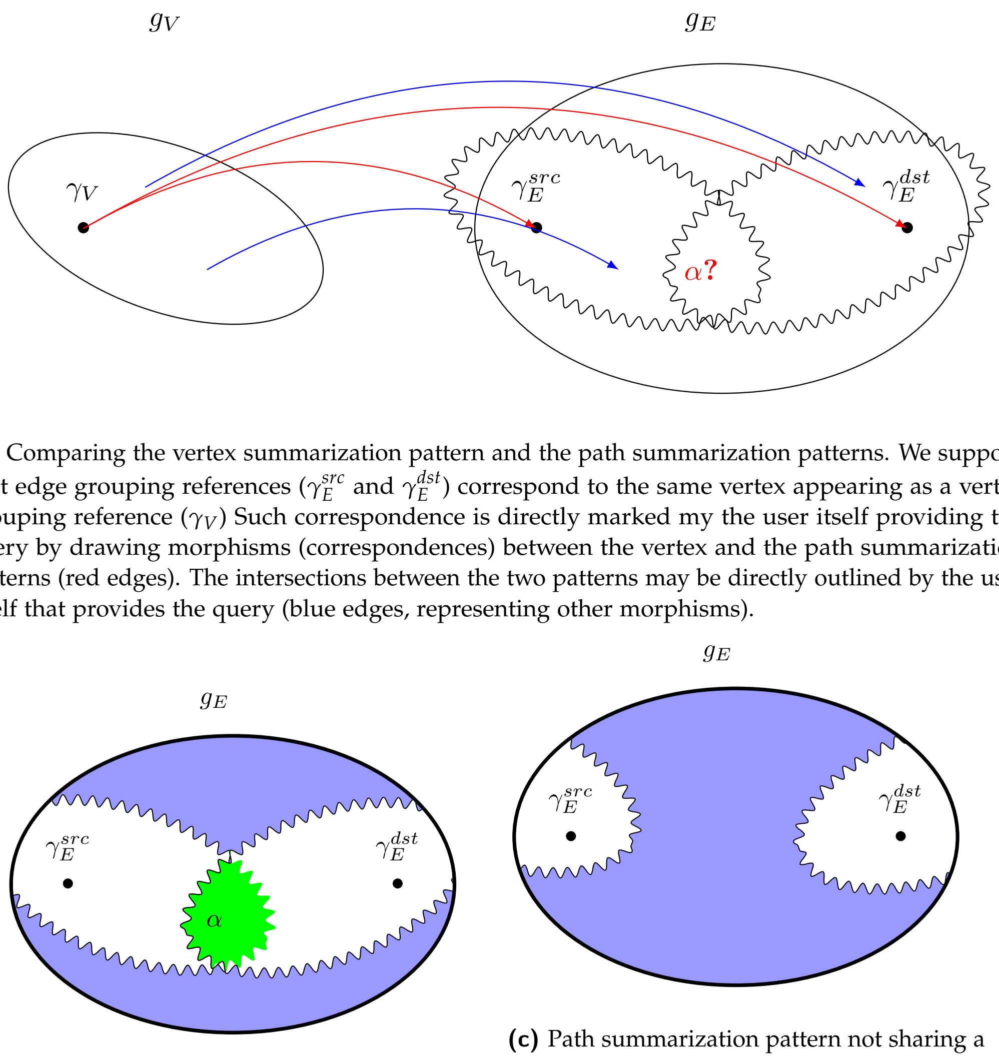
Before defining the general graph nesting operator and its THoSP algorithm, this seection detects the broader class of vertex and path summarization patterns optimizable as discussed in the chapter's introduction. The generation of the collections is only relevant with respect to actual data that is going to be nested and, in our case, we can only nest a subgraph of the graph resulting from the graph visiting process: consequently, within each pattern we must remark which elements are going to be nested at the final result. Instead of traversing a graph, generating the collections to be nested within the graph and check which one of them can be actually nested, we can directly create the nested graph while visiting the property graph.

First, we must provide a formal characterization of the grouping reference: we want to elect a subgraph γ_p for each graph pattern g_p such that each graph generated by the morphisms[3] $m_{g_p}(\eta)$ expose unique elements referring to γ_p. These grouping references elect the subcomponents identifying an entity over which the aggregation will be performed during the graph matching process. Hereby, we can provide the following formal definition for grouping references:

▸ **Definition 67** (Grouping reference). *Given a graph pattern g_p generating a set of morphisms $m_{g_p}(\eta)$ over a nested graph η, a **grouping reference** γ_p is a subpattern $\gamma_p \subseteq g_p$ restricting the possible morphisms generated by $m_{g_p}(\eta)$ to the f_i such that $|f_i(o)| = 1$ for each object $o \in O_{\gamma_p}$ and that another morphism $f_j \neq f_i$ is such that $f_i(O_{\gamma_p}) \neq f_j(O_{\gamma_p})$.* ◂

If we reduce the grouping references to one single vertex for vertex summarization patterns, and to two (distinct) vertices for path summarization patterns, we may reduce the computational complexity of aggregating the grouping reference. As already sketched by the introduction, the class of the desired nesting algorithms create new nested edges only over vertices that will be matched as grouping references and then nested. Moreover, we can choose to mark with a specific ξ value (e.g. "toNest") each pattern vertex and pattern edge to explicitely state which elements have to be nested in the final result; this implies that User Defined Functions are directly required to create the matched vertices and edges because they can be directly represented within the graph patterns.

[3] See Definition 18 on page 77 for a back reference.



(a) Comparing the vertex summarization pattern and the path summarization patterns. We suppose that edge grouping references (γ_E^{src} and γ_E^{dst}) correspond to the same vertex appearing as a vertex grouping reference (γ_V) Such correspondence is directly marked my the user itself providing the query by drawing morphisms (correspondences) between the vertex and the path summarization patterns (red edges). The intersections between the two patterns may be directly outlined by the user itself that provides the query (blue edges, representing other morphisms).

(b) Path summarization pattern sharing an α area of common patterns shared between the patterns, which are necessairly not the edge grouping references by definition.

(c) Path summarization pattern not sharing a common α area, although γ_E^{src} and γ_E^{dst} must always be present in g_E by hypothesis. This constraint guarantees that the newly created edge will be associated to a nested vertex originating from the vertex summarization pattern.

Figure 7.5 Vertex (V) and Path (E) summarization patterns for the query expressed in Example 35 on page 195. Vertex and edge grouping references are marked by a light blue circled node. As we can see, the vertex grouping reference depicts the same property expressed by edge grouping references.

Algorithm II.8 Grouping Reference Optimizable Queries (GROQ)

```
 1: new $\eta' := (\mathfrak{g}_{c+1}, O, \ell, \xi, \phi)$
 2:
 3: procedure GROQ($(g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \eta$):        ▷ $\eta = (\mathfrak{g}_c, O, \ell, \xi, \phi)$
 4:     $\alpha := g_V \cap g_E \backslash (\gamma_V \cup \gamma_E)$;
 5:     $\mathrm{lV} := []$;
 6:     if $\alpha \neq \varnothing$ then
 7:         for each graph $g^i$ generated from $m_\alpha(\eta)$ do
 8:             $\mathrm{lV} := \{f_i \in m_{g_V;\gamma_V}(\eta) | f_i(\alpha) = g^i\}$
 9:             GROQ$\alpha((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \mathrm{lV}, \eta)$
10:     else
11:         $\mathrm{lV} := m_{g_V;\gamma_V}(\eta)$
12:         GROQ$\alpha((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \mathrm{lV}, \eta)$
13:
14: procedure GROQ$\alpha((g_V, \gamma_V), (g_E, \gamma_E^{src}, \gamma_E^{dst}), m;\ \mathrm{lV}, \eta)$
15:     for each morphism $f_i \in \mathrm{lV}$ do
16:         $\{i_c\} := f_i(\gamma_V)$
17:         $\ell(i_{c+1}) := \ell(i_c)$; $\xi(i_{c+1}) := \xi(i_c)$; $\phi(i_{c+1}) = \phi(i_{c+1})|_{\mathrm{dom}(\phi(i_{c+1}))\backslash\{\text{"Entity"},\text{"Relationship"}\}}$
18:         $\phi(\mathfrak{g}_{c+1}, \text{"Entity"}) := \phi(\mathfrak{g}_{c+1}, \text{"Entity"}) \cup \{i_{c+1}\}$
19:         $\phi(i_{c+1}, \text{"Entity"}) := \phi(i_{c+1}, \text{"Entity"}) \cup \left\{ f_i(o) \,\middle|\, o \in O_{g_V}, \text{"toNest"} \in \xi(o) \wedge o \in \phi(o_{g_V}, \text{"Entity"}) \right\}$
20:         $\phi(i_{c+1}, \text{"Relationship"}) := \phi(i_{c+1}, \text{"Relationship"}) \cup \left\{ f_i(o) \,\middle|\, o \in O_{g_V}, \text{"toNest"} \in \xi(o) \wedge o \in \phi(o_{g_V}, \text{"Relationship"}) \right\}$
21:     for each morphism $f_i, f_j \in \mathrm{lV}$ do
22:         $\mathrm{lE} := \{f_k \in m_{g_E;\gamma_E^{src},\gamma_E^{dst}}(\eta) | f_i(\gamma_V) = f_k(\gamma_E^{src}), f_j(\gamma_V) = f_k(\gamma_E^{dst})\}$
23:         for each morphism $f_k \in \mathrm{lE}$ do
24:             $\{s_c\} := f_i(\gamma_E^{src})$;     $\{d_c\} := f_i(\gamma_E^{dst})$
25:             $\omega := \max O.\eta + dt(s, d)$
26:             $\ell(\omega_{c+1}) := \ell(s_c) \cup \ell(d_c)$; $\xi(i_{c+1}) := \xi(i_c) \cup \ell(d_c)$
27:             $\phi(\omega_{c+1}, \text{"Relationship"}) := \phi(\omega_{c+1}, \text{"Relationship"}) \cup f_k(\gamma_V)$
28:             $\phi(\omega_{c+1}, \text{"src"}) := \{s_{c+1}\}$;     $\phi(\omega_{c+1}, \text{"dst"}) := \{d_{c+1}\}$
29:             $\phi(\omega_{c+1}, \text{"Entity"}) := \phi(\omega_{c+1}, \text{"Entity"}) \cup \left\{ f_k(o) \,\middle|\, o \in O_{g_E}, \text{"toNest"} \in \xi(o) \wedge o \in \phi(o_{g_E}, \text{"Entity"}) \right\}$
30:             $\phi(\omega_{c+1}, \text{"Relationship"}) := \phi(\omega_{c+1}, \text{"Relationship"}) \cup \left\{ f_k(o) \,\middle|\, o \in O_{g_E}, \text{"toNest"} \in \xi(o) \wedge o \in \phi(o_{g_E}, \text{"Relationship"}) \right\}$
```

Figure 7.5 on the previous page provides an example on how graph nesting queries based on grouping references can be optimized for both vertex (g_V) and path (g_E) summarization queries; given that the users are going to provide both the vertex and the path summarization queries, such users must directly draw the correspondences between vertex and edge pattern queries, so that the correspondences can be promptly be identified by the query plan which can better optimize the whole query execution (a). After doing so, we can start to perform the general graph visiting algorithm for graph nesting (Algorithm II.8) by detecting which regions of both patterns are shared together in $\alpha = g_V \cap g_E$ (Figure 7.5b). Given that path summarization patterns' grouping references have distinct source and destination vertices by definition, source and destination vertices may not be represented in α (line 4). Consequently, we can first perform pattern matching over the input graph over α, thus allowing a partial instantiation of the g_V and g_E patterns, and then iteratively extend the nesting information after each visit of α and its own refinements. In particular, we can perform the algorithm as follows:

- Given a graphs extraction language m (not necessarily) supporting grouping references, we extract all the subgraphs g^i of η generated by morphisms $m_\alpha(\eta)$, when α is not empty (line 7). If α is otherwise an empty pattern, we must necessarily perform a complete visit of the vertex patterns g_V, and perform complete instantiations of such patterns (line 11).
- Given that the nested graph representation relies on the GSM model, we can iteratively construct the nested graph without knowing the complete information by relying on the ids of the expected elements, and we can provide the greatest subgraph of g matching

α after visiting each possible α matching result, represented as a morphism f_i. For this reason, the GROQα subroutine may be called in both cases.

- After providing a partial instantiation of the vertex summarization patterns via α, we find a vertex i_c matching the grouping reference γ_V to which we are going to nest the remaining objects: from i_c we generate a newly derived vertex i_{c+1} (line 16) preserving all the labels, expressions and containments of i_c (except from "Relationship"and "Entity"– line 17). In particular, the nesting content of i_{c+1} derives from the partial instantiation of the morphism f_i, by choosing the vertices and edges in ɳ which corresponds to vertex summarization objects marked with "toNest" (lines 19 and 20).
- At this point we can use the same semi-instantiated morphisms in 1V from α to partially instantiate the path summarization pattern, that is now going to be fully traversed (line 22). For each of these f_i instantiations, new edges are going to be generated, inheriting the labels, values and containments (except from "Relationship"and "Entity") from the matched edges grouping references, s_c and d_c. In particular, we can directly create associate to such edge the soruces and the targets represented by nested vertices, which will respectively be s_{c+1} and d_{c+1}.
- The procedure is iterated until the whole graph is not visited via subsequent morphisms, and hence all the matched elements are associated from the objects i_{c+1} (either vertices or edges) generated from the ones matched by the grouping reference i_c.

As we can see from the algorithm, the advantage of this approach is that the graph g^i and the instantiated morphisms (as a consequence of the graph matching phase) are promptly used to define the nested information (e.g., lines 18-20). It is evident that the aforementioned algorithm provides the best performances when γ_E^{src} and γ_E^{dst} are separated by one edge distance in α and both g_V and g_E create graph collections that are partitions of ɳ. On the other hand, this class of algorithms was already discussed in literature and, consequently, an approach describing how to optimize such scenarios can be already found in literature [JPR17]. Nevertheless, this chapter focuses on another types of algorithms, which are the ones where α contains two edges and one vertex; this class of problems, to the best of our knowledge, has not been discussed yet in current literature with respect to their optimizations. Please note that, when $\alpha = \varnothing$, the computational complexity of the algorithm may easily become quadratic ($|\phi(\mathrm{g}, \text{“Entity”})| + |\phi(\mathrm{g}, \text{“Entity”})|^2$).

7.3 Nested Graphs

In this chapter we try to define the nested graph data model independently from the GSM and GSQL query language. This choice will result in a limitation within the definition of the data structure and on the operators' formalization. We now define the *nested (property) graph database* from scratch as the following extension of the property graph data model for nested information:

▸ **Definition 68** (Nested Graph DataBase). *Given a set Σ^* of strings, a **nested (property) graph database** G is a tuple $G = \langle \mathcal{V}, \mathcal{E}, \lambda, \ell, \omega, \nu, \epsilon \rangle$, where $\mathcal{V}$ and $\mathcal{E}$ are disjoint sets, respectively referring to vertex and edge identifiers $o \equiv i_c \in \mathbb{N}$; c is an incremental unique number associated to each graph as in the GSM model.*

A function $\lambda \colon \mathcal{E} \to \mathcal{V}^2$ maps each edge to its source and target vertex. Each vertex and edge is assigned to multiple possible labels through the labelling function $\ell : \mathcal{V} \cup \mathcal{E} \to \wp(\Sigma^*)$. ω is a function mapping each vertex and edge into a relational tuple.

In addition to the previous components defining a property graph, we also introduce functions representing vertex members $\nu\colon (\mathcal{V} \cup \mathcal{E}) \to \wp(\mathcal{V})$ and edge members $\epsilon\colon (\mathcal{V} \cup \mathcal{E}) \to \wp(\mathcal{E})$. These functions induce the nesting by associating a set of vertices or edges to each vertex and edge. Each vertex or edge $o \in V \cup E$ induces a ***nested (property) graph*** *as the following pair:*

$$G_o = \left\langle \nu(o), \left\{ e \in \epsilon(o) \,\middle|\, \lambda(e) \in (\cup_{n \geq 0} \nu\epsilon^{(n)}(\{o\}))^2 \right\} \right\rangle$$

where $\nu\epsilon$ returns the vertices contained in both vertices and edges ($\nu\epsilon(x) = \nu(x) \cup \nu(\epsilon(x))$). We denote $f(X) := \bigcup_{x \in X} f(x)$ when $X \subseteq \operatorname{dom}(f)$. . ◂

As we previously observer, nested graphs can be also implemented in the GSM model. Since the member functions ν and ϵ induce the expansion of each single vertex or edge to a graph, we must avoid recursive nesting to support expanding operations. Therefore, we additionally introduce the following constraints to be set at a nested property graph database level:

▸ **Axiom 2** (Recursion Constraints)**.** *For each correctly nested property graph, each vertex $v \in \mathcal{V}$ must not contain v at any level of containment of ν and, any of its descendants m must not contain v:*

$$\forall v \in \mathcal{V}. \forall m \in \nu^+(v).\ \ m \neq v \land v \notin \cup_{n \geq 1} \nu\epsilon^{(n)}(m)$$

Similarly to vertices, any edge shall not contain itself at any nesting level:

$$\forall e \in \mathcal{E}. \forall m \in \epsilon^+(e). m \neq e \land e \notin \cup_{n \geq 1} \epsilon\nu^{(n)}(m)$$

where $\epsilon\nu$ returns the edges contained in both vertices and edges ($\epsilon\nu(x) = \epsilon(x) \cup \epsilon(\nu(x))$)

Please also note that this model has more restrictive constraints than the ones in the GSM model. This is due to the fact that GSM nested graphs differentiates vertices and edges by containing axioms while, in this case, we must restrict the edges to only the ones that are contained within the strongly nested components of the single nested graph. Nonetheless, a vertex v having a non-empty vertex or edge members is called **nested vertex**, while vertices with no members are simply referred to **simple vertices**. For edges, we respectively use the terms **nested edges** and **simple edges**.

▸ **Example 37.** *The property graph in Figure 7.1a can be represented by the graph $G_{(11_0)}$, which is a nested vertex contained in the following nested graph database:*

$$G = \langle \{0_0, 1_0, \dots, 5_0, 11_0\}, \{6_0, \dots, 10_0\}, \lambda, \ell, \omega, \nu, \epsilon \rangle$$

The nested vertex (11_0) represents a "Bibliography" graph ($\ell(11_0) = [$"Bibliography"$]$), to which an empty tuple is associated ($\omega(11_0) = \{\}$). Its vertex (ν) and edge (ϵ) members are defined as follows:

$$\nu(11_0) = \{0_0, \dots, 5_0\} \quad \epsilon(11_0) = \{6_0, \dots, 10_0\}$$

The simple edge 6 within the property graph in Figure 7.1a ($\nu(6_0) = \epsilon(6_0) = \varnothing$) has now id (6_0); it has one label, $\ell(6_0) = [$"AuthorOf"$]$, and it is associated to an empty tuple ($\omega(6_0) = \{\}$). The source and target vertices are $\lambda(6_0) = \langle 0_0, 3_0 \rangle$. Similar considerations can be carried out for each remaining edge.

The simple vertex 0 in the same Figure has id (0_0) in the present example; such vertex refers to the "Author" `Abigail Conner`*. This information is represented as follows:*

$$\ell(0_0) = [\text{"Author"}] \quad \nu(0_0) = \epsilon(0_0) = \varnothing$$

$$\omega(0_0) = \{\textbf{name}\colon \texttt{Abigail}, \textbf{surname}\colon \texttt{Conner}\}$$

Similar considerations can be carried out for each remaining vertex.

7.4 Graph Nesting

The graph nesting operator uses a classifier function grouping all the vertices and edges that shall appear as a member of a cluster C.

▸ **Definition 69** (Nested Graph Classifier, g_κ). *Given a set of cluster labels $\mathcal{C}$, a **nested graph classifier** function g_κ maps a nested graph G_o into a nested graph collection $\{G_C\}_{C\in\mathcal{C}, G_C\neq\emptyset}$ of subgraphs of G_o. Such function uses a classifier function $\kappa\colon \mathcal{V}\cup\mathcal{E}\to\wp(\mathcal{C})$ mapping each vertex or edge in either no graph or at least one non-empty subgraph. Each nested graph G_C is a pair $G_C = \langle\mathcal{V}_C, \mathcal{E}_C\rangle$ where $\mathcal{V}_C$ (and $\mathcal{E}_C$) is the set of all the vertices v (and edges e) in G_o having $C\in\kappa(v)$ (and $C\in\kappa(e)$). Therefore, the nested graph classifier is defined as follows:*

$$g_\kappa(G_o) = \{\, \langle\mathcal{V}_C, \mathcal{E}_C\rangle \mid C\in\mathcal{C}, (\mathcal{V}_C \neq \emptyset \vee \mathcal{E}_C = \emptyset)\,\}$$

◂

The former definition is also going to express graph pattern evaluations, where κ may be represented as a graph (cf. Neo4J). This assumption allows us to use the graphs in Figure 7.2 as possible κ. When κ is a graph, we denote as $\kappa \xrightarrow{f_C} G_C$ the function f_C associating each vertex (and edge) in κ to possibly more than one vertex (and one edge) in a subgraph $G_C \in g_\kappa(G_o)$. In order to represent the latter subgraphs as either vertices and edges, we may use the following User-Defined Functions:

▸ **Definition 70** (User-Defined Functions). *An **object user defined function** μ_Ω maps each subgraph $G_C\in g_\kappa(G_o)$ into a pair $\mu_\Omega(G_C) = (L, t)$, where $L\in\wp(\Sigma^*)$ is a set of labels and t is a relational tuple.*

*An **edge user defined function** μ_E maps each subgraph $G_C\in g_\kappa(G_o)$ into a pair of identifiers $\mu_E(G_C) = (s, t)$ where $s, t\in\mathbb{N}$.* ◂

▸ **Example 38.** *Within our use case scenario, μ_Ω must associate the authors' informations to each nested vertex resulting from $g_V(G_o)$, and create nested edges with* coAuthorship *label and no associated tuple:*

$$\mu_\Omega(G_C) = \begin{cases} ([\text{“coAuthorship”}],\ \emptyset) & G_C\in g_E(G_o) \\ (\ell(f_C(\gamma_V))),\ \omega(f_C(\gamma_V))) & G_C\in g_V(G_o) \end{cases}$$

While μ_Ω may be used for transforming subgraphs to both vertices and edges, μ_E is only used to map subgraphs to edges. In order to complete such transformation, we have to map each graph in $g_\kappa(G)$ into a new id $\mathbf{i_c}\notin\mathcal{V}\cup\mathcal{E}$, for which an indexing function ι_G over each G_C has to be defined within our specific task. As we will see in the next section, our scenario provides some constraints on both patterns; this allows the definition of an indexing function uniquely associating each matched subgraph G_C to the grouping references' ids. The previous functions are involved in the definition of our general graph nesting operator:

▸ **Definition 71** (Graph Nesting). *Given a nested graph G_{i_c} within a nested graph database G, an object user defined function μ_Ω, an edge user defined function μ_E and an indexing function ι_G, the graph nesting operator $\eta^{\mathbf{keep}}_{g_V,g_E,\mu_\Omega,\mu_E,\iota_G}$ converts each subgraph in $G_C\in g_V(G_{i_c})$ (and $G_C\in g_E(G_{i_c})$) into a nested vertex (and nested edge) $\iota_G(G_C)$ and adds them in a newly-created nested vertex; vertices and edges in G_{i_c} appearing neither in a nested vertex nor in a nested edge may be also* *returned if **keep** is set to* `true`*. This operator returns the following nested graph:*

$$\begin{aligned}\eta^{\boldsymbol{keep}}_{g_V,g_E,\mu_\Omega,\mu_E,\iota_G}(G_{i_c}) &= G_{i_{\overline{c}}} = \\ &= \Big\langle \{v \in \nu(i_c) | V(v) = \varnothing \wedge \boldsymbol{keep}\} \cup \iota_G(g_V(G_{i_c})), \\ &\qquad \{e \in \epsilon(i_c) | E(e) = \varnothing \wedge \boldsymbol{keep}\} \cup \iota_G(g_E(G_{i_c}))\Big\rangle\end{aligned}$$

where $\overline{c} = \max\{c | (i_c) \in \mathcal{V} \cup \mathcal{E}\} + 1$*. As a side effect of the graph nesting operation, the nesting graph database is updated using the nested graph classifier and user defined functions as follows:*

$$\begin{aligned}&\Big\langle \mathcal{V} \cup \iota_G(g_V(G_{i_c})) \cup \{(\overline{c}, i)\}, \quad \mathcal{E} \cup \iota_G(g_E(G_{i_c})), \\ &\lambda \oplus \bigoplus_{G_C \in g_E(G_{i_c})} \iota_G(G_C) \mapsto \mu_E(G_C), \\ &\ell \oplus \bigoplus_{G_C \in g_E(G_{i_c}) \cup g_V(G_{i_c})} \iota_G(G_C) \mapsto \mathsf{fst}\ \mu_\Omega(G_C), \\ &\omega \oplus \bigoplus_{G_C \in g_E(G_{i_c}) \cup g_V(G_{i_c})} \iota_G(G_C) \mapsto \mathsf{snd}\ \mu_\Omega(G_C), \\ &\nu \oplus \bigoplus_{G_C \in g_E(G_{i_c}) \cup g_V(G_{i_c})} \iota_G(G_C) \mapsto \mathcal{V}_C \\ &\quad \oplus dtl(i)_{\overline{c}} \mapsto \{v \in \nu(i_c) | V(v) = \varnothing \wedge \boldsymbol{keep}\} \cup \iota_G(g_V(G_{i_c})), \\ &\epsilon \oplus \bigoplus_{G_C \in g_E(G_{i_c}) \cup g_V(G_{i_c})} \iota_G(G_C) \mapsto \mathcal{E}_C \\ &\quad \oplus dtl(i)_{\overline{c}} \mapsto \{e \in \epsilon(i_c) | E(e) = \varnothing \wedge \boldsymbol{keep}\} \cup \iota_G(g_E(G_{i_c}))\Big\rangle\end{aligned}$$

where $(f \oplus g)(x)$ *returns* $g(x)$ *if* $x \in \mathrm{dom}(g)$ *and* $f(x)$ *otherwise, and both* f *and* g *are finite domain functions.* $a \mapsto b$ *denotes a finite function, which domain contains only* a*.* ◂

The following example describes the outcome of the graph nesting process.

▸ **Example 39.** *Figure 7.1b provides the result of* η *when the non-traversed vertices and edges are not preserved (**keep*** = `false`*) and where* V *and* E *are the ones represented in Figure 7.1. As showed by the former definition, the nesting operation updates the nested graph database by creating new nested vertices* $(0_1, 1_1, 2_1)$ *and nested edges* $(3_1, 5_1, 7_1, 8_1)$*. Such nested components are contained within the returned nested graph* G_{11_1}*, which is represented as a nested vertex with the following members:*

$$\nu(11_1) = \{0_1, 1_1, 2_1\} \quad \epsilon(11_1) = \{3_1, 5_1, 7_1, 8_1\}$$

The nested graph database updated as a side effect of the graph nesting may be represented as follows:

$$\begin{aligned}G' = \Big\langle &\{0_0, 1_0, \dots, 5_0, 11_0, 0_1, 1_1, 2_1, 11_1\}, \\ &\{6_0, \dots, 10_0, 3_1, 5_1, 7_1, 8_1\}, \\ &\lambda', \ell', \omega', \nu', \epsilon'\Big\rangle\end{aligned}$$

Let us now focus on the nested vertices and edges of G_{11_1}*. As requested by the UDF functions, each resulting nested* Author*(2) preserves the original vertices' tuple information, and its vertex members correspond to the* Papers *authored by the corresponding* Author*(1). For easing the* *nested graph representation, we assume that each* Author(2) *has an associated id* $(1, i)$*, which derives from a simple vertex with id* $(0, i)$ *in* $G_{(0,11)}$*. Therefore, vertex* 0_1 *is represented as follows:*

$$\ell'(0_1) = [\text{“Author”}] \quad \nu'(0_1) = \{3_0\} \quad \epsilon'(0_1) = \varnothing$$

$$\omega'(0_1) = \{\textbf{name}{:}\,\texttt{Abigail}, \textbf{surname}{:}\,\texttt{Conner}\}$$

Last, each resulting nested edge coAuthorship *has a* “coAuthorship” *label, it has no tuple information and its vertex members correspond to the* Papers *coauthored by source and target* Paper*. For easing the nested graph representation, we assume that each* coAuthorship *edge* $a_1 \to a'_1$ *has an associated id* $(\sum_{k=0}^{a+a'} k + a')_1$*, which derives from the grouping references. Therefore, edge* $0 \to 2$ *in Figure 7.1b is represented as follows:*

$$\ell'(5_1) = [\text{“coAuthorship”}]\; \nu'(5_1) = \{(3_0)\}\; \epsilon'(0_1) = \varnothing\; \omega'(0_1) = \{\}$$

In particular, we can freely assume that our nested graph pattern matching semantics s' acts as an UDF function, and hence associates to each graph cluster matched by g_E a source and a target vertex. On the other hand, while the previous formal definition of the graph nesting operator provides a general definition matching with Algorithm II.7 on page 202, the following algorithms allow to match the class of graph nesting optimizable problems that is going to be defined in the next section.

7.4.1 Two HOp Separated Patterns Algorithm

We now want to focus on a specific instance of the problem stated in Algorithm II.8: suppose to store a graph using adjacency lists similarly to the one proposed in the Graph Join algorithm chapter (Section 4.4.1 on page 101); in particular, the previous data structure is now extended with both vertex and edge containment, plus with both ingoing and outgoing edges for each single graph vertex. The latter requirement is added in order to satisfy the possibility to visit the edges backwards, thus allowing to navigate the graph in each possible direction. Given that the data structure requires a simple linear visit of the graph, no additional primary and secondary data structures are required. Nevertheless, during our serialization phase we provide both a primary index for accessing external informations (*VertexIndex*) and the serialization of all the vertices' adjacency lists, which is going to be used for traversing the graph (*VertexVals*).

The THoSP algorithm requires a preliminary phase, where the operand is loaded into secondary memory using the **input data representation**, and where primary and secondary indices are serialized for backward compatibility with graph joins. Such representation is presented in Figure 7.6: it is an extension of the usual graph adjacency lists (OutgoingEdges[], IngoingEdges[]) where each vertex o has an associated **Header** containing its id (o), its associated hash and the offset pointing to other serialized fields, such as the labelset and eventually its property-value representation ($\langle \ell(o), \omega(o) \rangle$). Last, for each edge we store its id and hash value, as well as the hash and the id of the adjacent vertex. Hash values are used within the proposed THoSP algorithm to store the correspondences with the graph patterns in Figure 7.2; therefore, each $\ell(o)$ is associated to a distinct hash value $h(o)$. We also suppose that the input graph data to be serialized does not represent an exact adjacency list: for this reason, the graph is firstly created in primary memory without the offset information, and then serialized into secondary memory.

In order to solve our specific graph nesting problem as presented in Example 35 on page 197, we have to formally determine the η parameters representing THoSP. We focus

Algorithm II.9 Two HOp Separated Patterns Algorithm (THoSP)

```
 1: procedure doNest($Index$, pattern, $f$, $\overrightarrow{memb} = \{m_1, \dots, m_n\}$)
 2:     for each $m_i \in \overrightarrow{memb}$ s.t. pattern($\overrightarrow{memb}$).doSerialize($m_i$) do
 3:         $Index$.write($\langle f, m_i \rangle$)
 4:
 5: procedure $\eta^{\texttt{false}}_{g_V, g_E, \dots}$(ɳ)
 6:     File $AdjFile$ = Open_MemoryMap(ɳ);                          ▷ Serialized operand
 7:     File $Nesting$ = Open(new);                                  ▷ $\nu \cup \epsilon$, member information
 8:     Adjacency $toSerialize$ =
           new Map<Vertex,<Edge,Vertex»();                          ▷ Nested graph, adj. list
 9:     $\alpha := V \cap E \backslash (\gamma_V \cup \gamma_E^{src} \cup \gamma_E^{dst})$;             ▷ Shared pattern
10:     for each vertex $v'$ in $AdjFile$ do                         ▷ $v' := v_c$
11:         if $v' \vDash \alpha$ then
12:             for each $(u', e, v') \vDash V$ do                   ▷ $u' := u_{c'}$
13:                 $\overline{u} := dtl(u)_{\overline{c}}$                            ▷ $\iota_G$, nested vertex
14:                 doNest($Nesting$, V, $\overline{u}$, $\{u', e, v'\}$)
15:                 for each $(w', e, v') \vDash V$ do               ▷ $w' := w_{c''}$
16:                     if $(u', e, v', e', w') \vDash E$ then
17:                         $\overline{w} := dtl(w)_{\overline{c}}$                    ▷ $\iota_G$, nested vertex
18:                         $\overline{e} := dtl(u, w)_{\overline{c}}$                 ▷ $\iota_G$, nested edge
19:                         doNest($Nesting$, E, $\overline{e}$, $\{u', e, v', e', w'\}$)
20:                         $toSerialize$.put($\overline{u}$, $\langle \overline{e}, \overline{w} \rangle$)
21:     $AdjFile$.serialize($toSerialize$);
22:     return ($AdjFile$, $Nesting$)                                ▷ Nested graph
```

on vertex (and edge) summarization patterns which grouping references associate unique vertices to distinct matching subgraphs; they require that:

$$\forall G_C \in g_V(G_o). \neg \exists G_d \in g_V(G_o).\ G_c \neq G_d \wedge f_C(\gamma_V) = f_D(\gamma_V)$$

$$\begin{aligned} \forall G_C \in g_E(G_o). \neg \exists G_d \in g_E(G_o).\ & G_c \neq G_d \\ & \wedge f_C(\gamma_E^{src}) = f_D(\gamma_E^{src}) \wedge f_C(\gamma_E^{dst}) = f_D(\gamma_E^{dst}) \end{aligned}$$

This requirement leads to a one-to-one mapping between subgraphs G_C and vertices matched by vertex (or edge) grouping references, that can be expressed by the following indexing function:

$$\iota_G(G_C) = \begin{cases} dtl(\mathsf{snd} f_C(\gamma_V))_{\overline{c}} & G_C \in g_V(G_o) \\ dtl\left(\mathsf{snd} f_C(\gamma_E^{src}),\ \mathsf{snd} f_C(\gamma_E^{dst})\right)_{\overline{c}} & G_C \in g_E(G_o) \end{cases}$$

This assumption permits a deterministic μ_E function, associating to each newly created nested edge from G_C two nested vertices having $f_C(\gamma_E^{src})$ and $f_C(\gamma_E^{dst})$ as vertex grouping references:

$$\mu_E(G_C) = \left\langle dtl(\mathsf{snd} f_C(\gamma_E^{src}))_{\overline{c}},\ dt(\mathsf{snd} f_C(\gamma_E^{dst}))_{\overline{c}} \right\rangle$$

Please note that the former function provides such association without additional join costs.

Algorithm II.9 provides the desired interpretation for the two pattern matching graphs returning the desired nested graph. After opening the previously-loaded graph operand through memory mapping (Line 6), we must first identify a sub-pattern α (Line 9) that is going to be visited only once within the graph (Line 11), after which either the vertex

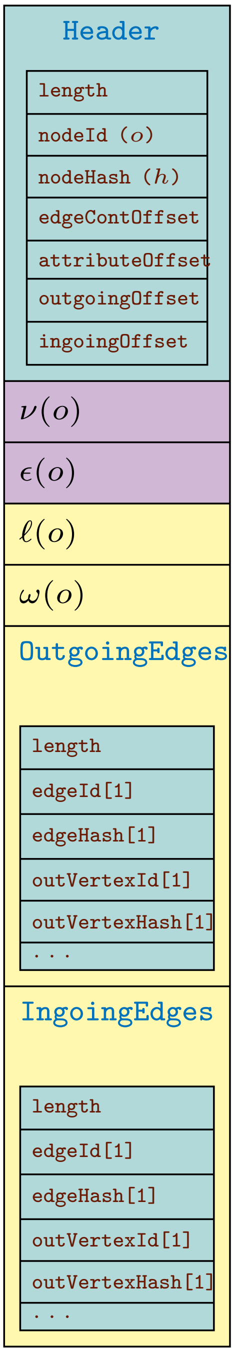


Figure 7.6 Extending the serialized graph data structure presented for graph join for the nesting operation. In particular, the present data structure extends each vertex representation in *VertexVals* (Figure 4.6b on page 103) in order to fully supports the nested graph data model: entities and relationships may now be contained into another data node (either a vertex or an edge). The first block of the serialized data structure contains the pointers towards the memory regions containing data which may vary in size. The fuchsia nodes remark the memory spaces where such data containments may be stored. Moreover, ingoing edges are stored as well as outgoing edges.

Operands' Vertices	Matched Graphs	**General Nesting** (ms)	**THoSP** (ms)
10	3	0.57	0.11
10^2	58	0.73	0.14
10^3	968	2.78	0.46
10^4	8,683	152.11	4.07
10^5	88,885	14,015.00	43.81
10^6	902,020	1,579,190.00	563.02
10^7	8,991,417	>1H	8,202.93
10^8	89,146,891	>1H	91,834.20

Table 7.1 Comparing the performances of the THoSP algorithm with the naive General Nesting algorithm. This comparison shows that the previously defined algorithm has a worse performance than the THoSP one.

or the path summarization pattern can be visited in their entirety. We also perform some restrictions over these patterns enhancing such optimizations: for each vertex v' matched by α (Line 11) we know that we must (possibly) visit all the edges going from v' towards the vertices γ_E^{src} and γ_E^{dst}. Therefore, having an edge as a constraint in α linking v towards γ_E^{src} or γ_E^{dst} both in E and V reduces the graph visiting time to the actual edges traversed from v' meeting the grouping references (Line 16). Therefore, we know when we finish our patterns' instantiation after exhaustively matching all the elements within the pattern. As a consequence, a "path join" is performed between the two nested patterns (Line 10 with 15): this is evident from the two vertex nested for loops appearing in the algorithm.

Our physical data model differentiates the *input data representation* from the **query result** (Line 22). We suppose that the latter is only used by the user to read the outcome of the nesting process as in other query languages (such as SPARQL and SQL) and does not have to produce "materialized views". Therefore, the result of the graph query itself can postpone the creation of a complete "materialized view", which will later use the same representation of the input data by using both the id information and the application of the User Defined Functions. In particular, the former dt function is used to associate both the nested vertices, $\overline{u}$ and $\overline{w}$, and the nested edge $\overline{e}$ to their grouping references, thus allowing to easily go back to the original grouping references by using the inverse function of dt, thus allowing the postponed application of the user defined functions.

Last, the doNest procedure performs the association between the nested vertices (and edges) f and its members within the input graph m_i. When the pattern requires that m_i should be a member of f in the final nested graph, doNest stores in a *Nesting* file those membership associations as pairs $\langle f, m_i \rangle$. By doing so, we omit the GROUP BY cost which affects the previously seen query languages.

Please note that if in g_E there is no path connecting α to γ_E^{src} or γ_E^{dst}, the problem may quickly become cubic with respect to the size of the vertices, because we must create all the possible permutations where v' is present alongside another element matching γ_E^{src} or γ_E^{dst}.

Table 7.1 provides a comparison between the general Nesting Algorithm sketched in Section 7.1.2 on page 201 and over the THoSP implementation of the query provided in our running example, under the assumptions that are going to be soon introduced in the next section. In particular, while THoSP increases linearly alongside the data size, the general nesting algorithm grows quadratically, thus quickly leading to a intractable time evaluation for big data scenarios. Hereby, the THoSP algorithm is going to be used in comparisons with other problem-specific queries on different query languages and data structures.

7.5 Experimental Evaluation

Through the following experiments we want to show that our approach outperforms the same proposed coauthorship nesting scenario on top of graph, relational, or document oriented databases. Therefore, we consider the time required to *(i)* serialize our data structure and *(ii)* evaluate the query plan. In the former we compare the loading and indexing times (the time required to store and index the data structure), and in the latter we time the query over the previously-loaded operand. This twofold analysis is required because, in some cases, the costly creation of several indices may lead to a better query performance.

The lack of ancillary data attached to either vertices or edges ($\forall o.\omega(o) = \{\}$) allows a better comparison of query evaluation times, which are now independent from the values' representations and more tailored to evaluate both the access time required for traversing the loaded operator and returning the nested representation. For our evaluations we choose a bibliography graph where vertices are only represented by vertex ids and label, and edges are represented only by both their label, and the source and target vertices' id. Such graph was generated by the gMark generator [BBC+17]: a Zipf's Law distribution with parameter 2.5 is associated to the ingoing distribution of each AUTHOROF, while a normal distribution between 0 and 80 is associated to its outgoing distribution. Each vertex represents either an AUTHOR or an authored PAPER having distinct ids. The resulting graph is represented as a list of triplets: source id, edge label (author of) and target id. The generator was configured to generate 8 experiments by incrementally creating a graph with vertices with a power of 10, that is from 10 to 10^8.

We performed our tests over a Lenovo ThinkPad P51 with a 3.00 GHz (until 4.00 GHz) Intel Xeon processor and 64 GB of RAM at 2.400 MHz. The tests were performed over a ferromagnetic Hard Disk at 5400 RPM with an NTFS File System. Given that the secondary memory representation is a simple extension of the one used for nested graphs, we assume that our data serialization is always outperforming with respect to graph libraries as discussed in Subsection 4.4.2.1 on page 105 for graph joins. Therefore, we only evaluate THoSP using the two pattern matching queries provided in the running example. As in Subsection 4.4.2.2 on page 107, we used default configurations for **Neo4J 3.3.0**, **PostgreSQL 9.6.6** and **ArangoDB 3.2.7**, while we changed the cache buffer configurations for **Virtuoso 7.2** (as suggested in the configuration file) for 64 GB of RAM; we also kept default multithreaded query execution plan. PostgreSQL queries were evaluated through the `psql` client and benchmarked using both `explain analyse` and `\timing` commands; the former allows to analyse SQL's query plans. Virtuoso was benchmarked through the Redland RDF library using directly the `librdf_model_query_execute` function; SPARQL's associated query plan was analysed via Virtuoso's `explain` statement. AQL queries over ArangoDB were evaluated directly through the `arangosh` client and benchmarked using the `getExtra()` method; statements' `explain` method was used to analyse AQL's query plans. Cypher queries were evaluated using the Java API through the `execute` method of a `GraphDatabaseService` object; the `EXPLAIN` statement was used to analyse the query's associated query plan. Given that only binary database connections were used (e.g., no HTTP), all the aforementioned conditions do not degrade the query evaluations. Last, given that all databases (except from Neo4J) was coded in C/C++ and that Neo4J provided the worst overall performances, we implemented serialization and THoSP only in C++.

Operands Size	**Operand Loading Time (C/C++)** (ms)				
Vertices ($\lvert V \rvert$)	PostgreSQL	Virtuoso	ArangoDB	Neo4J (Java)	**Nested Graphs** (C++)
10	8	3.67	43	3,951	0.23
10^2	18	6.86	267	4,124	0.65
10^3	45	23.53	1,285	5,256	5.54
10^4	225	371.40	11,478	11,251	39.14
10^5	1,877	3,510.96	135,595	1,193,492	376.07
10^6	19,076	34,636.80	1,362,734	>1H	4,016.06
10^7	184,421	364,129.00	>1H	>1H	47,452.10
10^8	1,982,393	>1H	>1H	>1H	527,326.00

(a) *Operand Loading and Indexing Time.* PostgreSQL and Neo4J have transactions, while Virtuoso and ArangoDB are transactionless. Nested Graphs are our proposed method which is transactionless.

Operands Size		**Two HOp Separated Pattern Time (C/C++)** (ms)				
Vertices ($\lvert V \rvert$)	Matched Graphs ($\lvert m_V(\eta)\rvert + \lvert m_E(\eta)\rvert$)	PostgreSQL	Virtuoso	ArangoDB	Neo4J (Java)	THoSP (C++)
10	3	2.10	11	15.00	681.40	0.11
10^2	58	9.68	63	3.89	1,943.98	0.14
10^3	968	17.96	63	12.34	>1H	0.46
10^4	8,683	69.27	364	46.74	>1H	4.07
10^5	88,885	294.23	4,153	508.87	>1H	43.81
10^6	902,020	2,611.48	50,341	7,212.19	>1H	563.02
10^7	8,991,417	25,666.14	672,273	922,590.00	>1H	8,202.93
10^8	89,146,891	396,523.88	>1H	>1H	>1H	91,834.20

(b) *Graph Nesting Time.* PLease note that the Graph Join Running Time. Each data management system is grouped by its graph query language implementation. This table clearly shows that the definition of our query plan clearly outperforms the default query plan implemented over those different graph query languages and databases.

Within the relational model the graph operand's edge information were stored in one single relational table. Similar approaches are automatically used in Virtuoso for representing RDF triple stores over its relational engine. As opposed to our implementation, all the current databases do not serialize the resulting nested graph in secondary memory.

At first, we must discuss the loading and indexing time (Table 7.2a). We shall compare Virtuoso and PostgreSQL first, because they are both based on a traditional relational database engine using one single table to store a graph. Virtuoso stores an RDF graph using its default format, while in PostgreSQL the graph was stored as described in Section 2.2. Given that Virtuoso is transactionless, it performed better at loading and index data for very small data sets (from 10 to 10^3) while, afterwards, the triple indexing time takes over on the overall performances. On the other hand, ArangoDB has not a relational data representation, and it serializes the data as JSON objects to which several external indices. Given that the only data loaded into ArangoDB are the edges' labels, all the time required to store the data is the indexing time. Neo4J's serialization proves to be inefficient, mainly because there are no constraints for data duplication and we must always check if the to-be-inserted vertex already exists. As a result, Neo4J's adoption of inverted indices from Lucene proves not to be useful at dynamically indexing graph data. Finally, our nested graph data structure creates adjacency lists directly when serializing the data, while primary indices are not used by our input data serialization, because the adjacency lists information is sufficient to join the edges in a two hop distance scenario.

Let us now consider the graph nesting time (Table 7.2b): albeit no specific triplet or key are associated to the stored graph, PostgreSQL appears to be more performant than Virtuoso on graph nesting. Please also note that the Virtuoso query engine rewrites the SPARQL query into SQL and, hereby, two SQL queries were performed in both cases. Since both data were represented in a similar way in secondary memory, the completely different performance between the two databases must be attributed to an inefficient rewriting of the SPARQL query into SQL. In particular, the nested representation using JSON array for PostgreSQL proved to be more efficient than returning a full RDF graph represented as triplets, thus arguing in favour of document stores. The PostgreSQL's efficiency is attributable to the run-time indexing time of the relational tables, that is shared with ArangoDB, where the indices are created at loading time instead: in both cases a single join operation is performed, plus some (either runtime or stored) index access time. Both PostgreSQL and ArangoDB use `GROUP BY`-s to create collections of nested values, separately for both vertices and edges. As observed in the previous paragraph, no primary index is used while performing the THoSP query, and adjacency graphs are returned using the same data structure used for graph joins: one single vertex is returned alongside the set of outgoing edges. Moreover, the nesting result is not created by using `GROUP BY`-s, but by sparsely creating an index that associates the container to its members: as a result, our query plan does not generate an additional cost for sorting and collecting all the elements because the nesting is provided during the graph traversal phase. Thus, the choice of representing the nesting information as a separate index proves to be more efficient.

7.6 Conclusions

To the best of our knowledge, this chapter proposed for the first time an algorithm (THoSP) which adds structural aggregation to an input graph. The final outcome of this process is a nested graph, which contains vertices and edges that may contain subgraphs of the original input graph. Such result is obtained by jointly visiting two graph patterns, the vertex and the pattern summarization, respectively leading to the creation of nested vertices and nested edges. The reason why such algorithm outperforms equivalent implementations over graph, relational and document based competitors is twofold: first, while their query plans force one graph visit per pattern, our solution allows to visit such graph only once; last, by detaching the graph representation from the membership information in the *query result* we can avoid the cost of performing an additional `GROUP BY` operation. This paper also provides a nested graph data model, allowing the definition of a generic graph nesting operator.

This solution was possible due to the assumptions derived from both GSM and GSQL, where it is showed that it is possible to refer any time to the elements that are going to be created later on within the computation, by simply deterministically knowing which is the id belonging to the element that is going to be created. We already formulated such assumption in our initial work on hypergraphs [Ber14]; this chapter proved the practical feasibility of this approach. Therefore, this chapter proves that the representation of nested graph may lead to the solution of current graph querying problems in a tractable way. Nevertheless, we believe that further studies will have to be done on the class of GROQ problems, thus extending our work on THoSP.

This chapter walked in the footsteps of current (graph) database literature, where data operations are defined at the single data structure level and not at the database level. As a consequence, the graph nesting operator must be represented as the creation of a new graph, requiring the update of the whole database as a side effect. On the other hand, all the operations that are performed on the GSM model via GSQL directly operate at the database level and provide the outcome of the nesting process as the final reference object. This chapter showed that the GSM data model provided a more clear and compact definition of the graph nesting operator (see Definition 60 on page 175).

Last, this chapter (alongside with the former) outlined the definition of a possible nesting operator permitting the representation of $\nu_{\cong}$, which is the last operator required by the LAV/GAV data integration approach. Hereby, the definition of the graph nesting operator accomplishes our task of providing the full set of data integration operators via paNGRAm and GSQL. We believe that further studies will have to be done to implement and test the GROQ algorithm over this specific data model.

Part III

Conclusions

8 Conclusions

"Cuius rei demonstrationem mirabilem sane detexi hanc marginis exiguitas non caperet."

— Pierre de Fermat

We remind our reader that each chapter provided a conclusion section, where we provided the intended future works for each thesis topic.

This thesis introduced the Generalized Semistructured Model, a data model allowing the representation of both graphs and nested data. This has led to the definition of an intermediate data model, **nested graphs**, which allows both to have a data structure with two main object classes (vertices and edges), and the data to be nested. However, this thesis has not treated other data models, such as RDF, where there are at least three classes of objects (note that some RDF properties may act as both vertices and edges). This requires the definition of ad hoc operators for this additional data model. Similar considerations may be adopted for hypergraphs, too. Therefore, we leave the definition of these operators to future developments in our research through the usage of **GSQL** over GSM data.

In particular, GSQL supported the definition of both **graph joins** and **graph nestings** that are required operations within the context of data integration: while the former operation allowed to chain the matching vertices and creates new edges by using an user-defined **es** semantics, the latter operation is required in the "blocking" operations where **(i)** the original obects' pieces of information are preserved and when **(ii)** we may reduce the data representation at its coarsest representation level. We also saw that the implementation of both graph operators revealed the deficiencies of current graph query languages in providing those operations, either because of their data model or because of their query plan. These limitations demand for a new query language allowing these two (nested) graph operations in an efficient implementation.

Last, we showed that GSM allowed the data integration between different data representations and GSQL may be used to express both data integration and data mining tasks, thus showing that such naïve query language may be used like an assembly interface towards which we can express different possible query languages. Further work has still to be carried out at the GSQL optimization level: equivalence axioms and new aggregated operators allowing computational enhancement frequently occurring in data querying tasks (similar to the composition $\bowtie$ and σ_θ providing the $\bowtie_\theta$ operator) are still to be provided. Moreover, most interesting graph features may arise whether GSQL may deal with graph data uncertainty [GT07] and graph metrics [DMR16]: further work should also analyse the ability of such language to manipulate data alongside with uncertainty measures.

A Resolving Alignments and Morphisms: OCaml Source Code

This appendix provides the code of the reconciliation of the matchings between hub schema and data via the data's source schema, provided as an outcome of the visit of the alignments (between the source schema and the hub schema) and of the morphisms (associating the data to its source schema). In order to do so, the GSM data model is also provided, alongside with some required utility functions. The following procedure describes the more theoretical concepts presented in Example 27 on page 150.

```
(** Utility function, treating ls as sets. It removes duplicated instances *)
let unique ls = List.sort_uniq compare ls

(** Unility function, flattening and treating ls as sets *)
let uFlat ls = unique (List.flatten ls)

(**
   Returns an equivalence class between the elements of outer and inner. The most
   common case is having inner and outer as the same set.

   outer = Set over which the class are created
   inner = Set over which the class are created from the elements of inner
   g = function combining the classes in outer and the inner elements matching the class
   f = equivalence between the elements and the class

   { { g(x,{ y \in inner | f(x,y)}) } | x \in outer }
*)
let toClass outer inner g f =
  unique (List.map (fun x -> g x (List.filter (fun y -> f x y) inner)) outer)

(** More handy that a toClass function, where there is one single set over which perform
    the classification *)
let group_by (ff : 'a -> 'b) (g : 'a -> 'c) (ll : 'a list) :
  ('b, 'c list) Hashtbl.t =
  List.fold_left
    (fun acc e ->
       let grp = ff e in
       let grp_mems = try Hashtbl.find acc grp with | Not_found -> []
       in (Hashtbl.replace acc grp ((g e) :: grp_mems); acc))
    (Hashtbl.create 100) ll

(** Return the group_by result within a list *)
let list_group_by f g ll =
  Hashtbl.fold (fun k v acc -> (k, v) :: acc) (group_by f g ll) []

let pair a b = (a, b)

let linject x = [ x ]

let limax def = function | [] -> def | x :: xs -> List.fold_left max x xs

let lomin = function | [] -> None | x :: xs -> Some (List.fold_left min x xs)

let lomax = function | [] -> None | x :: xs -> Some (List.fold_left max x xs)

let rec range m mm =
  if m == mm then [ m ] else if m > mm then [] else m :: (range (m + 1) mm)

let is_some = function | Some _ -> true | _ -> false

let get = function | Some a -> a | _ -> failwith "missing_value"

module IntSet =
  Set.Make(struct let compare = Pervasives.compare
                     type t = int
                      end)

(** Defining a function with finite domain *)
type ('a, 'b) funzione = { dom : 'a list; f : 'a -> 'b }
```

```
let emptyFun c = { dom = []; f = (fun x -> c); }

let domain f = f.dom

let expand f = f.f

(** Definition of a gsm as provided in the theorical framework *)
type gsm =
  { o : int; oO : int list; ell : (int, string list) funzione;
    xi : (int, string list) funzione;
    phi : (int, (string, int list) funzione) funzione
  }

let egsm reference =
  {
    o = reference;
    oO = [];
    ell = emptyFun [];
    xi = emptyFun [];
    phi = emptyFun (emptyFun []);
  }

(** Practical implementation, more handy to initialize, as provided by the
    Java implementation *)
type gsm_object =
  { id : int; e : string list; x : string list;
    p : (string * (int list)) list
  }

(* f + [i -> l] *)
let appendList i l ff =
  { dom = i :: ff.dom; f = (fun x -> if x = i then l else ff.f x); }

(* f + [i -> x -> l | (x,l) \in ls] *)
let cpListToFunction i ls fu =
  if (List.length ls) = 0
  then fu
  else
    {
      dom = i :: fu.dom;
      f =
        (fun j ->
           if j = i
           then
             List.fold_right (fun (s, il) ff -> appendList s il ff) ls
               (fu.f i)
           else fu.f j);
    }


(** Appends a gsm_object into the gsm theorical model *)
let appendElement (iexp : gsm_object) (gsm_elem : gsm) : gsm =
  {
    o = gsm_elem.o;
    oO = iexp.id :: gsm_elem.oO;
    ell = appendList iexp.id iexp.e gsm_elem.ell;
    xi = appendList iexp.id iexp.x gsm_elem.xi;
    phi = cpListToFunction iexp.id iexp.p gsm_elem.phi;
  }

(** Given the reference object ~reference creates a gsm from the ~gol list *)
let gsm_object_list_to_gsm (reference : int) (gol : gsm_object list) : gsm =
  List.fold_right (fun elem acc -> appendElement elem acc) gol
    (egsm reference)

(* \varphi_{gsm} *)
let varphi gsm =
  let www = gsm.phi
  in
    (cpListToFunction ()
       (List.map
          (fun x ->
             (x, (let yyy = www.f x in uFlat (List.map yyy.f yyy.dom))))
          www.dom)
       (emptyFun (emptyFun []))).
      f ()

(* \varphi^{step}_{gsm}(o) *)
```

```
let rec varphiRec (step : int) gsm o =
 if step = 0
 then [ o ]
 else
  if step = 1
  then (varphi gsm).f o
  else
   unique
    (List.flatten (List.map (varphi gsm).f (varphiRec (step - 1) gsm o)))

(** Returning the maximum height of the gsm, at any object *)
let overallGsmHeight gsm =
 let maxHeightO o =
  let rec inner ls acc =
   if (List.length ls) == 0
   then acc
   else inner (uFlat (List.map (varphi gsm).f ls)) (acc + 1) in
  let l = (varphi gsm).f o in if (List.length l) == 0 then 0 else inner l 1
 in limax 0 (List.map (fun o -> maxHeightO o) gsm.o0)

let varphiplus gsm o =
 uFlat
  (List.map (fun x -> varphiRec x gsm o) (range 1 (overallGsmHeight gsm)))

let varphistar gsm (o : int) =
 unique
  (o ::
   (uFlat
     (List.map (fun x -> varphiRec x gsm o)
       (range 1 (overallGsmHeight gsm)))))

(** returns the elements of gsm of which o is a content *)
let containerOf gsm o =
 let w = varphi gsm
 in
  unique
   (List.map fst
     (List.filter (fun (i, l) -> List.mem o l)
       (List.map (fun x -> (x, (w.f x))) w.dom)))

(** returns the elements of gsm of which o is a content, alongside with the label associated
    ↪ to it *)
let containerOfWithLabel g o =
 List.flatten
  (List.map
    (fun x ->
     let y = g.phi.f x
     in
      List.map fst
       (List.filter (fun ((x, y), l) -> List.mem o l)
          (List.map (fun z -> ((x, z), (y.f z))) y.dom)))
    g.phi.dom)

(** Returns the basic correspondences to which o is associated to *)
let cvBase g o =
 List.append (g.xi.f o) (List.map snd (containerOfWithLabel g o))

let cvBase2 g o = g.xi.f o

let cvRecAll cvBaseFun (step : int) g o =
 uFlat (List.map (cvBaseFun g) (varphistar g o))

let cvRec step g o = cvRecAll cvBase step g o

let cvRec2 step g o = cvRecAll cvBase2 step g o

(**
 *  After initializing the function with the following elements, it returns if the
 *  two elements match because of a shared common value
 *
 *  l = value extraction function to be applied on the left
 *  r = value extraction function to be applied on the right
 *  step = depth step on both elements to go in order to search the common values
 *  gl = gsm for the left elements
 *  gr = gsm for the right elements
 *  vartheta = binary predicate creating the equivalences
*)
let cvTest l r step gl gr vartheta o op =
  List.exists
    (fun a -> List.exists (fun ap -> vartheta a ap) (cvRecAll r step gr op))
    (cvRecAll l step gl o)

(* Function to be used when the data to be compared belong to the same type, and hence the
     ↪ parts where to extract the data are similar *)
let cvTestData vef step g vartheta = cvTest vef vef step g g vartheta

type correspondence = { src : int; dst : int }

let oe x y = { dst = x; src = y; }

type morphism = { schema : int; data : int list }

let mo x y = { schema = x; data = y; }

(** These are all the kinds of schema alignments that we can achieve on one single
     correspondence *)
type schema_alignments =
  { ell_corr : correspondence list; xi_onelements : correspondence list;
    xi_contents : correspondence list
  }

let sa x y z = { ell_corr = x; xi_onelements = y; xi_contents = z; }

(** These are all the possible data inputs over which we play on *)
type data_input =
  { hub_schema_gsm : gsm; source_schema_gsm : gsm; data_gsm : gsm
  }

let edata x y z =
  { hub_schema_gsm = egsm x; source_schema_gsm = egsm y; data_gsm = egsm z; }

type ell_corr = correspondence list

type xi_onelements = correspondence list

type xi_contents = correspondence list

type morphisms = morphism list

(** Filters the ell_corr having o as a target *)
let getEllO (ec : ell_corr) omega = List.filter (fun x -> x.dst = omega) ec

(** Filters the ell_onelements having o as a target *)
let getExprO (ec : xi_onelements) omega =
  List.filter (fun x -> x.dst = omega) ec

(** Returns the set of objects matched to the schema through schemaId *)
let getW (ml : morphisms) schemaId =
  uFlat
    (List.map (fun x -> x.data)
       (List.filter (fun x -> x.schema = schemaId) ml))

(* Returning the data elements matching with the ell definition *)
(** Returns the \mathcal{I} associated to the currently evaluated object *)
let callI (ec : schema_alignments) (ml : morphisms) omega =
  uFlat (List.map (fun x -> getW ml x.src) (getEllO ec.ell_corr omega))

(* Filtering the ell by the matchings with the xi values *)
(** Returns the set of objects matched to the schema through schemaId, where
    the source is kept distinct per morphism *)
let getWSplitted (ml : morphisms) schemaId =
  List.map (fun x -> x.data) (List.filter (fun x -> x.schema = schemaId) ml)

let eee (ec : xi_onelements) (ml : morphisms) omega =
  uFlat (List.map (fun x -> getWSplitted ml x.src) (getExprO ec omega))

(* Given a collection of n elements of ('a list list), it joins the elements together *)
let rec join ls =
  let rec joining ls acc =
    match ls with
    | a :: b ->
        if (List.length a) = 0
        then joining b acc
        else
          joining b
            (List.flatten
               (List.map (fun x -> List.map (fun y -> x @ y) a) acc))
    | [] -> acc
  in
    match ls with
    | [] -> []
    | [ a ] -> a
    | a :: b -> if (List.length a) = 0 then join b else joining b a

(* Given a collection of n elements of ('a list list), it creates the cross product *)
let rec cross ls =
  let rec crossing ls acc =
    match ls with
    | a :: b ->
        if (List.length a) = 0
        then crossing b acc
        else
          crossing b
            (List.flatten
               (List.map (fun x -> List.map (fun y -> x @ (linject y)) a) acc))
    | [] -> acc
  in
    match ls with
    | [] -> []
    | [ a ] -> if (List.length a) = 0 then [] else linject a
    | a :: b ->
        if (List.length a) = 0
        then cross b
        else crossing b (List.map linject a)

(* Same function as cross, but keeps the schema information as the left part
   when the lists preserve the information from where they came from *)
let rec pair_cross ls = ((List.map fst ls), (cross (List.map snd ls)))

(* Same function as cross, but it is used on the incoming values from the contents *)
let rec pair_cross2 ls =
  ((List.flatten (List.map fst ls)), (uFlat (cross (List.map snd ls))))

(**
 *
 * extractor = function used to select the right alignment from "sa"
 * sa        = alignments to be selected via extractor
 * ml        = morphisms going from the source schema to the data
 * omega     = element originating from the hub schema
 *)
let expandAnyMorphismOverObject2 extractor (sa : schema_alignments)
                                 (ml : morphisms) (omega : int) =
  (* I want to select all the xi mappings (over object) that have omega as a target
      srcw = { x.src | x\in (sa.extractor), x.dst = omega }
   *)
  let srcw =
    List.map (fun x -> x.src)
      (List.filter (fun x -> x.dst = omega) (extractor sa)) in
  (* I want to select all the morphisms that have their schema eleme tin srcw
      mlFilt = { x\in ml | x.schema \in srcw }
             = { y\in ml | x\in (sa.extractor), x.dst = omega, x.src = y.schema }
   *)
  let mlFilt = List.filter (fun x -> List.mem x.schema srcw) ml
  in
    (* performs a group by over the mlFilt by source schema elements*)
    pair_cross (list_group_by (fun x -> x.schema) (fun x -> x.data) mlFilt)


let expandXiMorphismsOverObject (sa : schema_alignments) (ml : morphisms)
                                (omega : int) =
  expandAnyMorphismOverObject2 (fun x -> x.xi_onelements) sa ml omega

let expandEllMorphismsOverObject (sa : schema_alignments) (ml : morphisms)
                                 (omega : int) =
  expandAnyMorphismOverObject2 (fun x -> x.ell_corr) sa ml omega

let rec listifte f =
  function | [] -> [] | a :: b -> (if f a then a else []) :: (listifte f b)

(** This function filters the ells that match with the correspondent \xi values
*)
let crossLists vef step vartheta (datei : data_input) (schema, ells)
               (schema2, xixs) =
  if (List.length ells) = 0
  then (schema2, xixs)
  else
    if (List.length xixs) = 0
    then (schema, ells)
    else
      (schema,
       (List.map
          (fun bigL ->
             listifte
               (fun l ->
                  List.exists
                    (fun bigE ->
                       List.exists
                         (fun e ->
                            List.for_all
                              (fun vare ->
                                 List.exists
                                   (fun lam ->
                                      cvTestData vef step datei.data_gsm
                                        vartheta vare lam)
                                   l)
                              e)
                         bigE)
                    xixs)
               bigL)
          ells))

let crossMorphisms vef step vartheta (datei : data_input)
                   (sa : schema_alignments) (ml : morphisms) (omega : int) =
  let (schema, ells) = expandEllMorphismsOverObject sa ml omega in
  let (schema2, xixs) = expandXiMorphismsOverObject sa ml omega
  in
    (* If I only have xi-matches and no \lambda ones, I return directly the data referenced
        ↪ by the source*)
    crossLists vef step vartheta datei (schema, ells) (schema2, xixs)

(** Definition of the relatve height *)
let rh (g : gsm) o op =
  let h = overallGsmHeight g in
  let r = range 1 h
  in
    match lomax
            (List.map (fun c -> c + 1)
               (List.filter (fun n -> List.mem op (varphiRec n g o)) r))
    with
    | Some n -> Some n
    | None ->
        (match lomax
                 (List.map (fun c -> c + 1)
                    (List.filter (fun n -> List.mem o (varphiRec n g op)) r))
         with
         | Some n -> Some (- n)
         | None ->
             if
               (o = op) ||
                 (List.exists
                    (fun opp ->
                       (List.mem o ((varphi g).f opp)) &&
                         (List.mem op ((varphi g).f opp)))
                    (List.filter (fun x -> (( != ) x o) || (( != ) x op))
                       g.oO))
             then Some 0
             else None)

let h g =
  limax 0
    (List.map
       (fun op ->
          match rh g g.o op with
          | Some n -> if n >= 0 then n else 0
          | None -> 0)
       (varphistar g g.o))

let ho gsm o curr =
  let ll =
    List.map (fun x -> rh gsm curr x)
      (List.filter (fun x -> ( != ) x curr) (varphistar gsm o))
  in limax 0 (List.map get (List.filter is_some ll))
```

```
let contentSort gsm o =
  List.sort (fun x y -> compare (- (ho gsm o x)) (- (ho gsm o y)))
    ((varphi gsm).f o)

(** Returns the indices indicating which elements has to be extracted to perform
 *  the object selection.
 *
 *  xischemas = elements over which extract the filtering
 *  ls = list providing the relevant elements to be extractes
 *  headers = elements originating from the lambda-matches from the hub schema,
 *            from which the hub schema elements are referenced
 *)
let filterIndicesFromContents xischemas ls headers =
  List.map fst
    (List.filter
       (fun (i, xsl) ->
          List.exists (fun x -> (x.src = xsl) && (List.mem xsl headers))
            xischemas)
       (List.mapi pair ls))

let rec
  postVisitPace visit vef step vartheta (datei : data_input)
                (sa : schema_alignments) (ml : morphisms) (omega : int) =
  (* Performs a postVisit on the contents. I choose the contents by their relative
   * height with respect to the containment relations in o
   *)
  let (schema, results) =
    pair_cross2
      (List.map (visit datei) (contentSort datei.hub_schema_gsm omega)) in
  (* Return all the lambda-matches associated to omega *)
  let currentElls = getEllO sa.ell_corr omega in
  (* contents' filtered schemas referencing only to the elements that truly have to be
       ↪ matched *)
  let indices =
    filterIndicesFromContents sa.xi_contents
      (varphistar datei.source_schema_gsm omega) schema in
  (* Filtering the second-xi-matches having as sources the source schema objects appearing
       ↪ within the results*)
  let (fs, fr) =
    ((List.map (List.nth schema) indices),
     (List.map (fun ls -> List.map (List.nth ls) indices) results))
  in
    crossLists vef step vartheta datei
      (crossMorphisms vef step vartheta datei sa ml omega) (fs, fr)

let performAssociations vef step vartheta datei sa ml =
  (** Memoizing the visit steps. This is possible due to postVisit *)
  let cache = Hashtbl.create (List.length datei.hub_schema_gsm.oO) in
  let rec postVisit omega =
    try Hashtbl.find cache omega
    with
    | Not_found ->
        (* Performs a postVisit on the contents. I choose the contents by their
         * relative height with respect to the containment relations in o.
         * schema = $\mathcal{S}_{o_i}$
         * results = $\mathcal{I}_{o_i}$
         *)
        let (schema, results) =
          pair_cross2
            (List.map postVisit (contentSort datei.hub_schema_gsm omega)) in
        (* Return all the lambda-matches associated to omega *)
        let currentElls = getEllO sa.ell_corr omega in
        (* contents' filtered schemas referencing only to the elements that
         * truly have to be matched *)
        let indices =
          filterIndicesFromContents sa.xi_contents
            (varphistar datei.source_schema_gsm omega) schema in
        (* Filtering the second-xi-matches having as sources the source schema
         * objects appearing within the results*)
        let (fs, fr) =
          ((List.map (List.nth schema) indices),
           (List.map (fun ls -> List.map (List.nth ls) indices) results)) in
        let f =
          crossLists vef step vartheta datei
            (crossMorphisms vef step vartheta datei sa ml omega) (fs, fr)
        in (Hashtbl.add cache omega f; f)
  in (postVisit datei.hub_schema_gsm.o; cache)
```

```
(* EXAMPLES *)
let ggg =
  gsm_object_list_to_gsm 0
    [ {
        id = 0;
        e = [ "ciao" ];
        x = [ "expr" ];
        p = [ ("left", [ 2 ]); ("right", [ 3; 5 ]) ];
      }; { id = 3; e = []; x = [ "expr" ]; p = [ ("left", [ 2 ]) ]; };
      { id = 2; e = [ "ciao" ]; x = []; p = [ ("right", [ 4 ]) ]; };
      { id = 5; e = [ "ciao" ]; x = [ "expr" ]; p = []; };
      { id = 4; e = [ "ciao" ]; x = [ "expr" ]; p = []; } ]

let xmlist = [ oe 0 100; oe 0 200; oe 0 300 ]

let lmlist = [ oe 0 400; oe 0 500 ]

let mlist =
  [ mo 100 [ 1; 2; 3; 4 ]; mo 100 [ 2; 3; 4; 5 ]; mo 200 [ 5; 6; 7 ];
    mo 200 [ 8; 9 ]; mo 300 [ 10; 11 ]; mo 400 [ 1; 2; 3; 303; 4 ];
    mo 400 [ 100; 279; 305; 303 ]; mo 400 [ 10 ]; mo 500 [ 8; 9 ] ]

let em = expandXiMorphismsOverObject (sa lmlist xmlist []) mlist 0

let lm = expandEllMorphismsOverObject (sa lmlist xmlist []) mlist 0

let cm =
  crossMorphisms (fun x y -> [ y ]) 0 (fun x y -> x = y) (edata 0 0 0)
    (sa lmlist xmlist []) mlist 0
```

B Dovetailing lemmas

▸ **Lemma 5.1.1.** $dt(x+1,y) = dt(x,y) + x + y + 1$

Proof. The proof can be carried out by simple rewriting. Given that:

$$dt(x,y) = \frac{(x+y)(x+y+1)}{2} + y$$

and consequently that:

$$dt(x+1,y) = \frac{(x+y+1)(x+y+2)}{2} + y$$

we have that

$$\begin{aligned} dt(x+1,y) - dt(x,y) &= \frac{(x+y+1)}{2}(\cancel{x+y} + 2 \cancel{-x-y}) \\ &= x + y + 1 \end{aligned}$$

Therefore:

$$dt(x+1,y) = dt(x,y) + x + y + 1$$

◂

▸ **Lemma 5.1.2.** $dt(x+i,y) = dt(x,y) + i(x+y) + \sum_{n=0}^{i} n$

Proof. We can prove it by induction over i.

$i = 0$ holds by substitution of i and reflexivity.
$i = 1$ holds by lemma 5.1.1.
$i = n+1$ The induction hypothesis is the following:

$$dt(x+n,y) = dt(x,y) + n(x+y) + \sum_{j=0}^{n} j$$

Therefore, we have that:

$$\begin{aligned} dt(x+n+1,y) &= dt(x+n,y) + x + y + (n+1) \\ &\overset{IH}{=} dt(x,y) + (n)(x+y) + \sum_{j=0}^{n} j + (x+y) + (n+1) \\ &= dt(x,y) + (n+1)(x+y) + \sum_{j=0}^{n+1} j \end{aligned}$$

◂

▸ **Lemma 5.1.3.** $dt(x,y+1) = dt(x,y) + x + y + 2$

Proof. The proof can be carried out by simple rewriting. Given that:

$$dt(x,y) = \frac{(x+y)(x+y+1)}{2} + y$$

and consequently that:

$$dt(x, y+1) = \frac{(x+y+1)(x+y+2)}{2} + y + 1$$

we have that

$$\begin{aligned} dt(x+1, y) - dt(x, y) &= 1 + \frac{(x+y+1)}{2}(\cancel{x+y} + 2 \cancel{-x-y}) \\ &= x + y + 2 \end{aligned}$$

Therefore:

$$dt(x+1, y) = dt(x, y) + x + y + 2$$

◂

C Expressing containment functions in `script`

This appendix provides the script notation for the mathematical notation used in the GSQL definitions. We chose to use the latter instead of `script` due to the fact that this notation is more compact and more readable. On the other hand, a `script` implementation of such functions shows how such definitions can be implemented in any system. Therefore, this appendix is going to provide the implementation of the aforementioned mathematical notation.

Script for $\psi_\cup$

The associated `script` expression can be defined as follows:

```
fold [o.phi,
      k-> { {{k[0][0][1], k[1][k[0][0][1]]++k[0][1]}} ++
            select(k[1] : y -> {not(y[0] == k[0][1])})
          },
      {}
     ]
```

Please remember that `k` is a pair, where `k[0]` is the current element of `o.phi` while `k[0][1]` is the accumulated value that, in our case, is the step-by-step reformulation of the containment.

Script for $\psi_\setminus$

Before providing the definition of $\psi_\setminus$ in `script`, we must define some script utility functions, such as some set (or, as in this case, list) operations.

```
difference = x -> select (x[0] : y -> {not(y in x[1])})
intersect = x -> select (x[0] : y -> { y in x[1]})
distinct = x -> {
        fold {x,
              y->{if (y[0] in y[1]) then y[1] else [y[0]]++y[1]},
              {}
             }
}
```

We can also define some further shorthands for accessing each object's attributes. Let us remember that for each element `x` in `o.phi`, `x[0]` represents the attribute a associated to the containment, and hence `x[0][0]` represent the original operand, while `x[0][1]` represent the attribute appearing in the original operand.

```
keys ≝ (distinct (map (o.phi :x ->x[0][1])))
operands ≝ (distinct (map (o.phi :x ->x[0][0])))
```

Moreover, the following function returns a list of pairs `{p, 1}` for every containment $\phi(g, [\mathtt{y},\mathtt{p}]) = 1$ found in the disjunct united nested graph belonging to the `y`-th operand.

```
getOperandList = y -> { map(select(o.phi : x -> {x[0][0] = y}) :
                            x -> {x[0][1], x[1]})
                      }
```

By combining some of the previous functions, we obtain the final desired result:

```
map(keys : x -> {{x,
                  (difference {(getOperandList 0)[x],
                               (getOperandList 1)[x]
                              })
                 }
                }
   )
```

Script for $\psi_\cap$

Similarly to the previous step, we have to perform the intersection over all the operands over the common set of attributes.

```
map(keys : x -> {
             (fold { remove operands[0] in operands,
                     y -> { (intersect {(getOperandList (y[0]))[x
                        ↪ ],
                                        y[1]
                                       })
                          },
                     (getOperandList (operands[0]))[x]
                   })
})
```

Bibliography

AAB$^+$17 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

ACB06 Paolo Atzeni, Paolo Cappellari, and Philip A. Bernstein. Model-independent schema and data translation. In *Proceedings of the 10th International Conference on Advances in Database Technology*, EDBT'06, pages 368–385, Berlin, Heidelberg, 2006. Springer-Verlag.

ACPT99 Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill, 1st edition, 1999.

ACPT09 Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Basi di dati. Modelli e linguaggi di interrogazione*. McGraw-Hill, Milan, 3rd edition, 2009.

ACZH10 Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 41–50, New York, NY, USA, 2010. ACM.

AG08 Renzo Angles and Claudio Gutierrez. The expressive power of sparql. pages 114–129. 2008.

AGG$^+$15 Julien Aligon, Enrico Gallinucci, Matteo Golfarelli, Patrick Marcel, and Stefano Rizzi. A collaborative filtering approach for recommending olap sessions. *Decision Support Systems*, 69:20 – 30, 2015.

AGK$^+$17 Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. Reasoning on data partitioning for single-round multi-join evaluation in massively parallel systems. *Commun. ACM*, 60(3):93–100, 2017.

Agr88 Rakesh Agrawal. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Trans. Softw. Eng.*, 14(7):879–885, July 1988.

AH11 Dean Allemang and James Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2011.

ATOR16 Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 431–446, New York, NY, USA, 2016. ACM.

Atr15 Medha Atre. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *SIGMOD Conference*, pages 1793–1808. ACM, 2015.

AU79 Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM.

AW04 Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *(e)Proceedings of the Thirtieth International Conference* *on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 336–347, 2004.

BAdCG16 Freddy Brasileiro, João Paulo A. Almeida, Victorio Albani de Carvalho, and Giancarlo Guizzardi. Expressive multi-level modeling for the semantic web. In *International Semantic Web Conference (1)*, volume 9981 of *Lecture Notes in Computer Science*, pages 53–69, 2016.

BBC$^+$15 Anders Berglund, Scott Boag, Donald D. Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. Xml path language (xpath) 3.1, January 2015.

BBC$^+$17 G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017.

BBMP15 Flavio Bertini, Giacomo Bergami, Danilo Montesi, and Paolo Pandolfi. Predicting frailty in elderly people using socio-clinical databases. *5th Workshop on Data Mining for Medicine and Healthcare*, 2015.

BBPV11 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Theory and practice of monotone minimal perfect hashing. *ACM Journal of Experimental Algorithmics*, 16, 2011.

BCC$^+$16 Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Martin Rezk, and Guohui Xiao. OBDA beyond relational dbs: A study for mongodb. In *Description Logics*, volume 1577 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.

BCM$^+$10 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, New York, NY, USA, 2nd edition, 2010.

BDIP$^+$13 Gioele Barabucci, Angelo Di Iorio, Silvio Peroni, Francesco Poggi, and Fabio Vitali. Annotations with earmark in practice: A fairy tale. In *Proceedings of the 1st International Workshop on Collaborative Annotations in Shared Environment: Metadata, Vocabularies and Techniques in the Digital Humanities*, DH-CASE '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

BDK$^+$13 Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 121–132, New York, NY, USA, 2013. ACM.

BELP07 Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. Graph markup language (GraphML). In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*. CRC Press, 2007.

Ber14 Giacomo Bergami. Hypergraph Mining for Social Networks . Master's thesis, Italy, 2014.

BFL13 Pablo Barceló, Gaelle Fontaine, and Anthony Widjaja Lin. *Expressive Path Queries on Graphs with Data*, pages 71–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

BG05 Alain Bretto and Luc Gillibert. Hypergraph-based image representation. In Luc Brun and Mario Vento, editors, *GbRPR*, volume 3434 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2005.

BHLS17 Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. *An Introduction to Description Logic*. Cambridge University Press, 1st edition, 2017.

BK14 Piotr Bródka and Przemyslaw Kazienko. Multilayered social networks. In *Encyclopedia of Social Network Analysis and Mining*, pages 998–1013. 2014.

BL11 Antonio Badia and Daniel Lemire. A call to arms: Revisiting database design. *SIGMOD Rec.*, 40(3):61–69, November 2011.

BLC^{+}17 Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Schema inference for massive JSON datasets. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.*, pages 222–233, 2017.

BLP04 Ulrik Brandes, Jürgen Lerner, and Christian Pich. Gxl to graphml and vice versa with xslt. In *International Workshop on Graph-Based Tools (GraBaTso)*, 2004.

BM08 Giacomo Buratti and Danilo Montesi. Ranking for approximated xquery full-text queries. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases, BNCOD 25, Cardiff, UK, July 7-10, 2008. Proceedings*, pages 165–176, 2008.

BMM16 Giacomo Bergami, Matteo Magnani, and Danilo Montesi. On joining graphs. abs/1608.05594, 2016.

BMM17 Giacomo Bergami, Matteo Magnani, and Danilo Montesi. A join operator for property graphs. In *Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017.*, 2017.

BN09 Jens Bleiholder and Felix Naumann. Data fusion. *ACM Comput. Surv.*, 41(1):1:1–1:41, January 2009.

Bra03 Thorsten Brants. Natural language processing in information retrieval. In *Computational Linguistics in the Netherlands 2003, December 19, Centre for Dutch Language and Speech, University of Antwerp*, 2003.

BS07 Ahmet Bulut and Ambuj K. Singh. Indexing and querying data streams. In *Data Streams - Models and Algorithms*, pages 237–259. 2007.

Cab98 Luca Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 13(IC982734):22–56, 1998.

CDD^{+}04 Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, pages 74–83, New York, NY, USA, 2004. ACM.

CG85 Stefano Ceri and Georg Gottlob. Translating sql into relational algebra: Optimization, semantics, and equivalence of sql queries. *IEEE Trans. Softw. Eng.*, 11(4):324–345, April 1985.

CGP00 Óscar Corcho and Asunción Gómez-Pérez. A roadmap to ontology specification languages. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, EKAW '00, pages 80–96, London, UK, UK, 2000. Springer-Verlag.

Che76 Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

CHoSU93 Text REtrieval Conference, D. K. Harman, National Institute of Standards, and Technology (U.S.). *The first Text REtrieval Conference (TREC-1) [mi-*

croform]. U.S. Dept. of Commerce, National Institute of Standards and Technology Gaithersburg, MD, 1993.

CJ10 Zheng Chen and Heng Ji. Graph-based clustering for computational linguistics: A survey. In *Proceedings of the 2010 Workshop on Graph-based Methods for Natural Language Processing*, TextGraphs-5, pages 1–9, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

CJQ16 Gong Cheng, Cheng Jin, and Yuzhong Qu. HIEDS: A generic and efficient approach to hierarchical dataset summarization. In *Procs. of IJCAI 2016*, pages 3705–3711, 2016.

CLNP06 T. Calders, L. V.S. Lakshmanan, R. T. Ng, and J. Paredaens. Expressive power of an algebra for data mining. *ACM Trans. on Database Systems*, 31(4):1169–1214, 2006.

CM90a Mariano P. Consens and Alberto O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.

CM90b Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in graphlog and datalog. In Serge Abiteboul and ParisC. Kanellakis, editors, *ICDT '90*, volume 470 of *Lecture Notes in Computer Science*, pages 379–394. Springer Berlin Heidelberg, 1990.

Cod71 E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

Cod90 E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

CYZ$^+$08 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. IEEE Computer Society, 2008.

DEGI10 Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Algorithms and theory of computation handbook. chapter Dynamic Graph Algorithms, pages 9–9. Chapman & Hall/CRC, 2010.

DGLL$^+$17 Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. *Using Ontologies for Semantic Data Integration*, pages 187–202. Springer International Publishing, Cham, 2017.

Dit16 Jens Dittrich. *Patterns in Data Management: A Flipped textbook*. Jens Dittrich, Saarland University, Germany, 1 edition, 2016.

dMDS$^+$14 Marie-Catherine de Marneffe, Timothy Dozat, Natalia Silveira, Katri Haverinen, Filip Ginter, Joakim Nivre, and Christopher D. Manning. Universal stanford dependencies: A cross-linguistic typology. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation, LREC 2014, Reykjavik, Iceland, May 26-31, 2014.*, pages 4585–4592, 2014.

DMR16 Mark E. Dickinson, Matteo Magnani, and Luca Rossi. *Multilayer Social Networks*. Cambridge University Press, 2016.

DSP$^+$14 Souripriya Das, Jagannathan Srinivasan, Matthew Perry, Eugene Inseok Chong, and Jayanta Banerjee. A tale of two graphs: Property graphs as RDF in oracle. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 762–773, 2014.

DSUBGVn$^+$10 D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 37–48, Berlin, Heidelberg, 2010. Springer-Verlag.

DVMT15 Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Approximate querying of rdf graphs via path alignment. *Distributed and Parallel Databases*, 33(4):555–581, 2015.

EALP$^+$15 Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

EM09 Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Semantic Web Information Management*, pages 501–519. Springer, 2009.

EN16 Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2016.

ES13 Jérôme Euzenat and Pavel Shvaiko. *Ontology matching*. Springer-Verlag, Heidelberg (DE), 2nd edition, 2013.

EV12a Lorena Etcheverry and Alejandro A. Vaisman. Enhancing olap analysis with web cubes. In Elena Simperl, Philipp Cimiano, Axel Polleres, Óscar Corcho, and Valentina Presutti, editors, *ESWC*, volume 7295 of *Lecture Notes in Computer Science*, pages 469–483. Springer, 2012.

EV12b Lorena Etcheverry and Alejandro A. Vaisman. Qb4olap: A vocabulary for olap cubes on the semantic web. In Juan Sequeda, Andreas Harth, and Olaf Hartig, editors, *COLD*, volume 905 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

EV12c Lorena Etcheverry and Alejandro A. Vaisman. Qb4olap: A vocabulary for olap cubes on the semantic web. In *COLD*, volume 905 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

FB09 George H.L. Fletcher and Peter W. Beck. Scalable indexing of rdf graphs for efficient join processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, pages 1513–1516, New York, NY, USA, 2009. ACM.

Fel98 Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

FLM$^+$12 Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6(3):313–338, 2012.

FPG15 Valeria Fionda, Giuseppe Pirrò, and Claudio Gutierrez. Nautilod: A formal language for the web of data graph. *TWEB*, 9(1):5:1–5:43, 2015.

GB10 Carolina Galleguillos and Serge Belongie. Context based object categorization: A critical survey. *Comput. Vis. Image Underst.*, 114(6):712–722, June 2010.

GCR$^+$17 Mikhail Galkin, Diego Collarana, Ignacio Traverso Ribón, Maria-Esther Vidal, and Sören Auer. Sjoin: A semantic join operator to integrate heterogeneous RDF graphs. In *Database and Expert Systems Applications - 28th*

International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I, pages 206–221, 2017.

GGK09 Stephen Gould, Tianshi Gao, and Daphne Koller. Region-based segmentation and object detection. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 655–663. Curran Associates, Inc., 2009.

GHKR11 Anika Groß, Michael Hartung, Toralf Kirsten, and Erhard Rahm. Mapping composition for matching large life science ontologies. In *ICBO*, volume 833 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

GHMP11 Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. Foundations of semantic web databases. *Journal of Computer and System Sciences*, 77(3):520 – 541, 2011. Database Theory.

GL02 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

GMP+12 Matteo Golfarelli, Federica Mandreoli, Wilma Penzo, Stefano Rizzi, and Elisa Turricchia. OLAP query reformulation in peer-to-peer data warehousing. *Inf. Syst.*, 37(5):393–411, 2012.

GMUW08 Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

GP13 Aldo Gangemi and Valentina Presutti. A multi-dimensional comparison of ontology design patterns for representing *n*-ary relations. In *SOFSEM*, volume 7741 of *Lecture Notes in Computer Science*, pages 86–105. Springer, 2013.

GPG14 Ben Goertzel, Cassio Pennachin, and Nil Geisweiller. *Engineering General Intelligence, Part 1 - A Path to Advanced AGI via Embodied Learning and Cognitive Synergy*, volume 5 of *Atlantis Thinking Machines*. Atlantis Press, 2014.

GRS+15 Amine Ghrab, Oscar Romero, Sabri Skhiri, Alejandro Vaisman, and Esteban Zimányi. *Advances in Databases and Information Systems: 19th East European Conference, ADBIS 2015, Poitiers, France, September 8-11, 2015, Proceedings*, chapter A Framework for Building OLAP Cubes on Graphs, pages 92–105. Springer International Publishing, Cham, 2015.

GRS+16 Amine Ghrab, Oscar Romero, Sabri Skhiri, Alejandro A. Vaisman, and Esteban Zimányi. Grad: On graph database modeling. *CoRR*, abs/1602.00503, 2016.

GS98 David Genest and Eric Salvat. A platform allowing typed nested graphs: How cogito became cogitant (research note). In *Conceptual Structures: Theory, Tools and Applications, 6th International Conference on Conceptual Structures, ICCS '98, Montpellier, France, August 10-12, 1998, Proceedings*, pages 154–164, 1998.

GT07 Lise Getoor and Ben Taskar. *Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2007.

GYQ+12 Jun Gao, Jeffrey Yu, Huida Qiu, Xiao Jiang, Tengjiao Wang, and Dongqing Yang. Holistic top-k simple shortest path join in graphs. *IEEE Trans. on Knowl. and Data Eng.*, 24(4):665–677, April 2012.

Har87 David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

HAR11 Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.

He07 Huahai He. *Querying and Mining Graph Databases*. PhD thesis, University of California at Santa Barbara, CA, USA, 2007. AAI3283657.

HG16 Jürgen Hölsch and Michael Grossniklaus. An algebra and equivalences to transform graph patterns in neo4j. *Fifth International Workshop on Querying Graph Structured Data*, 2016.

HGR13 Michael Hartung, Anika Groß, and Erhard Rahm. Composition methods for link discovery. In *BTW*, volume 214 of *LNI*, pages 261–277. GI, 2013.

HIK11 Richard Hammack, Wilfried Imrich, and Sandi Klavzar. *Handbook of Product Graphs, Second Edition*. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2011.

HL97 D.L. Hall and J. Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1), Jan 1997.

HP15 Olaf Hartig and Jorge Pérez. *The Semantic Web - ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, chapter LDQL: A Query Language for the Web of Linked Data, pages 73–91. Springer International Publishing, Cham, 2015.

HP17 Olaf Hartig and Jorge Pérez. An initial analysis of facebook's graphql language. *11th Alberto Mendelzon International Workshop on Foundation of Databases and the Web (AMW)*, 06 2017.

HRJM15 Sébastien Harispe, Sylvie Ranwez, Stefan Janaqi, and Jacky Montmain. Semantic similarity from natural language and ontology analysis. *Synthesis Lectures on Human Language Technologies*, 8, 2015.

HS12 Brian Henderson-Sellers. *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages.* Springer Briefs in Computer Science. Springer, 2012.

HSSW06 Richard C. Holt, Andy Schürr, Susan E. Sim, and Andreas Winter. Gxl: A graph-based standard exchange format for reengineering. *Sci. Comput. Program.*, 60(2):149–170, 2006.

IK00 Wilfred Imrich and Sandi Klavzar. *Product Graphs. Structure and Recognition*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

Inc14 Neo Technology Inc. *Cypher Cheat Scheet*, 2014.

IP07 Wilfried Imrich and Iztok Peterin. Recognizing cartesian products in linear time. *Discrete Mathematics*, 307(3-5):472–483, 2007.

IPPV14 Angelo Di Iorio, Silvio Peroni, Francesco Poggi, and Fabio Vitali. Dealing with structural patterns of XML documents. *JASIST*, 65(9):1884–1900, 2014.

JFL15 Wararat Jakawat, Cécile Favre, and Sabine Loudcher. OLAP Cube-based Graph Approach for Bibliographic Data. In *SOFSEM 2016*, Harrachov, Czech Republic, November 2015.

JKA$^+$17 Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, pages 3:1–3:8, 2017.

Joh11 Jeffrey Johnson. *Hypernetwors in the Science of Complex Systems*. Imperial College Press, London, UK, UK, 2011.

JPR17 Martin Junghanns, André Petermann, and Erhard Rahm. Distributed grouping of property graphs with gradoop. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, pages 103–122, 2017.

JPT+16 M. Junghanns, A. Petermann, N. Teichmann, K. Gomez, and E. Rahm. Analyzing extended property graphs with apache flink. *SIGMOD workshop on Network Data Analytics (NDA)*, 07 2016.

KÖ6 Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)*, 5(4):369–385, December 2006.

KB17 Sarah Kohail and Chris Biemann. Matching, re-ranking and scoring: Learning textual similarity by incorporating dependency graph alignment and coverage features. *18th International Conference on Computational Linguistics and Intelligent Text Processing.*, 2017.

Khu12 Udayan Khurana. An introduction totemporal graph data management. Technical report, Computer Science Department, University of Maryland, 2012.

KKKR13 Bahador Khaleghi, Alaa Khamis, Fakhreddine O. Karray, and Saiedeh N. Razavi. Multisensor data fusion: A review of the state-of-the-art. *Information Fusion*, 14(1):28 – 44, 2013.

Kos08 Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388, 2008.

KRRV15 Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. *The Semantic Web - ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I*, chapter SPARQL with Property Paths, pages 3–18. Springer International Publishing, Cham, 2015.

KW82 Casimir A. Kulikowski and Sholom M. Weiss. Representation of expert knolwedge for consultation: The CASNET and EXPERT proejcts. In *Artificial Intelligence in Medicine*, number 2. Westview Press, Boulder, Colorado, 1982.

Lar04 Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

LBO+15 Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4):479–514, Dec 2015.

Len02 Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246, 2002.

LJ14 Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84, 2014.

LLDM09 J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

LN07 Ulf Leser and Felix Naumann. *Informationsintegration*. dpunkt.verlag, 2007.

Löw93 Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theor. Comput. Sci.*, 109(1&2):181–224, 1993.

LS99 Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, February 1999.

LS16 Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

Lu06 James J. Lu. A data model for data integration. *Electron. Notes Theor. Comput. Sci.*, 150(2):3–19, March 2006.

Luh58 H. P. Luhn. A business intelligence system. *IBM J. Res. Dev.*, 2(4):314–319, October 1958.

LVJRT14 Juan Antonio Lossio-Ventura, Clement Jonquet, Mathieu Roche, and Maguelonne Teisseire. *Advances in Natural Language Processing: 9th International Conference on NLP, PolTAL 2014, Warsaw, Poland, September 17-19, 2014. Proceedings*, chapter Yet Another Ranking Function for Automatic Multiword Term Extraction, pages 52–64. Springer International Publishing, Cham, 2014.

LWZ04 Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Query languages and data models for database sequences and data streams. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 492–503, 2004.

LY05 Hong-Cheu Liu and Jeffery X. Yu. Algebraic equivalences of nested relational operators. *Inf. Syst.*, 30(3):167–204, May 2005.

LZ09 Ling Liu and M. Tamer Zsu. *Encyclopedia of Database Systems*. Springer Publishing Company, Incorporated, 1st edition, 2009.

LZ16 Jian Liu and X.X. Zhang. Dynamic labeling scheme for xml updates. *Know.-Based Syst.*, 106(C):135–149, August 2016.

MFK01 Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML queries on heterogeneous data sources. In *VLDB*, pages 241–250. Morgan Kaufmann, 2001.

MM06 Matteo Magnani and Danilo Montesi. A unified approach to structured and XML data modeling and manipulation. *Data Knowl. Eng.*, 59(1):25–62, 2006.

MM09 Matteo Magnani and Danilo Montesi. Towards relational schema uncertainty. In *Proceedings of the 3rd International Conference on Scalable Uncertainty Management*, SUM '09, pages 150–164, Berlin, Heidelberg, 2009. Springer-Verlag.

MM10 Matteo Magnani and Danilo Montesi. Us-sql: Managing uncertain schemata. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1195–1198, New York, NY, USA, 2010. ACM.

MRS08 Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

MSV17 József Marton, Gábor Szárnyas, and Dániel Varró. Formalising opencypher graph queries in relational algebra. *CoRR*, abs/1705.02844, 2017.

MZRA16 Emily K. Mallory, Ce Zhang, Christopher Ré, and Russ B. Altman. Large-scale extraction of gene interactions from full-text literature using deepdive. *Bioinformatics*, 32(1):106–113, 2016.

Neo13 The Neo4j Team NeoThechnology. The neo4j manual v2.0.0, 2013.

New10 Mark Newman. *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, USA, 2010.

NHNR17 Markus Nentwig, Michael Hartung, Axel-Cyrille Ngonga Ngomo, and Erhard Rahm. A survey of current link discovery frameworks. *Semantic Web*, 8(3):419–436, 2017.

NN92 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

NNH05 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

NP12a Roberto Navigli and Simone Paolo Ponzetto. Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artif. Intell.*, 193:217–250, December 2012.

NP12b Roberto Navigli and Simone Paolo Ponzetto. Multilingual WSD with just a few lines of code: the BabelNet API. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL 2012)*, Jeju, Korea, 2012.

NRA^{+}17 Sergi Nadal, Oscar Romero, Alberto Abelló, Panos Vassiliadis, and Stijn Vansummeren. An integration-oriented ontology to govern evolution in big data ecosystems. In *EDBT/ICDT Workshops*, volume 1810 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

Obj11 Object Management Group (OMG). Uml 2.4.1 superstructure specification, 2011.

Odi92 P. Odifreddi. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers (Studies in Logic and the Foundations of Mathematics)*. North Holland, new ed edition, February 1992.

omg96 Common Facilities RFP-5: Meta-Object Facility, cf/96-05-02, June 1996.

OMG11a OMG. *OMG MOF 2 XMI Mapping Specification, Version 2.4.1*. Object Management Group, August 2011.

OMG11b OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1*. Object Management Group, August 2011.

oRD12 IBM Journal of Research and Development. *This is Watson*, volume 56(3/4). IBM Co., May/July 2012.

PAG09 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. 34(3):16:1–16:45, September 2009.

PAK16 Minjae Park, Hyun Ahn, and Kwanghoon Pio Kim. Workflow-supported social networks: Discovery, analyses, and system. *Journal of Network and Computer Applications*, 75:355 – 373, 2016.

PAKR16 Thomas Palomares, Youssef Ahres, Juhana Kangaspunta, and Christopher Ré. Wikipedia knowledge graph with deepdive. In *Wiki, Papers from the 2016 ICWSM Workshop, Cologne, Germany, May 17, 2016*, 2016.

PG92 Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, 1992.

PH01 Alexandra Poulovassilis and Stefan G. Hild. Hyperlog: A graph-based system for database browsing, querying, and update. *IEEE Trans. Knowl. Data Eng.*, 13(2):316–333, 2001.

Pie02 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

PJMR14 André Petermann, Martin Junghanns, Robert Müller, and Erhard Rahm. Graph-based data integration and business intelligence with biiig. *Proc. VLDB Endow.*, 7(13):1577–1580, August 2014.

PL94 A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Trans. Information Systems*, 12(1):35–68, 1994.

PLB15 Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. Graphite: An extensible graph traversal framework for relational database management systems. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 29:1–29:12, New York, NY, USA, 2015. ACM.

Plu99 Detlef Plump. Term graph rewriting. In *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 3–61, 1999.

PMB$^+$17 André Petermann, Giovanni Micale, Giacomo Bergami, Martin Junghanns, Alfredo Pulvirenti, and Erhard Rahm. Scalable frequency mining and ranking of generalized multi-dimensional graph patterns. *ICDIM*, 2017. Forthcoming.

Pog06 Antonella Poggi. *Structured and Semistructured Data Integration*. PhD thesis, Università degli Studi di Roma "La Sapienza", Italy, 2006.

Pre10 Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.

PSAH16 Victor M. Parra, Ali Syed, Mohammad Azeem, and Malka N. Halgamuge. Pentaho and jaspersoft: A comparative study of business intelligence open source tools processing big data to evaluate performances. In *International Journal of Advanced Computer Science and Applications*, volume 7(10), pages 3–61, 2016.

PSF12 Peter F. Patel-Schneider and Enrico Franconi. Ontology constraints in incomplete and complete data. In *Proceedings of the 11th International Conference on The Semantic Web - Volume Part I*, ISWC'12, pages 444–459, Berlin, Heidelberg, 2012. Springer-Verlag.

PVG92 Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, March 1992.

QZY$^+$11 Qiang Qu, Feida Zhu, Xifeng Yan, Jiawei Han, Philip S. Yu, and Hongyan Li. *Efficient Topological OLAP on Information Networks*, pages 389–403. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

R9̈4 Thomas Rölleke. Equivalences of the probabilistic relational algebra. Technical report, 1994.

Rah16 Erhard Rahm. The case for holistic data integration. In *East European Conference on Advances in Databases and Information Systems*, pages 11–27. Springer, 2016.

Ren03 Arend Rensink. Model checking graph grammars. Technical report, Department of Computer Science, University of Twente, Netherlands, 2003.

RH08 Jennifer Rowley and Richard Hartley. *Organizing Knowledge: An Introduction to Managing Access to Information*. Ashgate Publishing, Ltd., 2008.

RHKB13 Alexander Richter, Julia Heidemann, Mathias Klier, and Sebastian Behrendt. Success measurement of enterprise social networks. *Wirtschaftsinformatik*, (20), 2013.

RJ16 Rahimeh Rouhi and Mehdi Jafari. Classification of benign and malignant breast tumors based on hybrid level set segmentation. *Expert Syst. Appl.*, 46(C):45–59, March 2016.

RJKK15 Rahimeh Rouhi, Mehdi Jafari, Shohreh Kasaei, and Peiman Keshavarzian. Benign and malignant breast tumors classification based on region growing and cnn segmentation. *Expert Syst. Appl.*, 42(3):990–1002, February 2015.

Rod15 Marko A. Rodriguez. The gremlin graph traversal machine and language. *CoRR*, abs/1508.03843, 2015.

Rol13 Mara Carina Roldn. *Pentaho Data Integration Beginner's Guide*. Packt Publishing, 2nd edition, 2013.

RPZ10 Yuan Ren, Jeff Z. Pan, and Yuting Zhao. Closed world reasoning for owl2 with nbox. *Tsinghua Science & Technology*, 15(6):692 – 701, 2010.

RRK$^+$08 Frederick Reiss, Sriram Raghavan, Rajasekar Krishnamurthy, Huaiyu Zhu, and Shivakumar Vaithyanathan. An algebraic approach to rule-based information extraction. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 933–942, Washington, DC, USA, 2008. IEEE Computer Society.

RW97 S. E. Robertson and S. Walker. On relevance weights with little relevance information. *SIGIR Forum*, 31(SI):16–24, July 1997.

RWE13 Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

Sar08 Sunita Sarawagi. Information extraction. *Found. Trends databases*, 1(3):261–377, March 2008.

SAZ11 Rania Soussi, Marie-Aude Aufaure, and Hajer Baazaoui Zghal. Graph database for collaborative communities. In *Community-Built Databases*, pages 205–234. Springer, 2011.

SCD16 Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD Conference*, pages 1961–1976. ACM, 2016.

SFS$^+$15 Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901, New York, NY, USA, 2015. ACM.

SGP08 Simon Schenk, Paul Gearon, and Alexandre Passant. Sparql 1.1 update. Technical report, W3C, 2008. Published online on October 14th, 2010 at http://www.w3.org/TR/2010/WD-sparql11-update-20101014/.

SHJ+13 Nagiza F. Samatova, William Hendrix, John Jenkins, Kanchana Padmanabhan, and Arpan Chakraborty. *Practical Graph Mining with R*. Chapman & Hall/CRC, 2013.

SHK+14 Min Song, Nam-Gi Han, Yong-Hwan Kim, Ying Ding, and Tamy Chambers. Correction: Discovering implicit entity relation with the gene-citation-gene network. *PLOS ONE*, 9(1), 01 2014.

Shm11 Florian Shmedding. Incremental sparql evaluation for query answering on linked data. In *Second International Workshop on Consuming Linked Data*, COLD2011, 2011.

Sin01 Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

SK06 M. Saeki and H. Kaiya. On Relationships Among Models, Meta Models, and Ontologies. In *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006.

SLL02 Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

SPG+07 Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51 – 53, 2007. Software Engineering and the Semantic Web.

SPR17 Alieh Saeedi, Eric Peukert, and Erhard Rahm. Comparative evaluation of distributed clustering schemes for multi-source entity resolution. *ADBIS*, 2017.

SS93 Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs - Discrete Mathematics for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Springer, 1993.

SSSF09 Rolf Sint, Stephanie Stroka, Sebastian Schaffert, and Roland Ferstl. Combining unstructured, fully structured and semi-structured information in semantic wikis. In *4th Semantic Wiki Workshop (SemWiki 2009) at the 6th European Semantic Web Conference (ESWC 2009), Hersonissos, Greece, June 1st, 2009. Proceedings.*, 2009.

THP08 Yuanyuan Tian, Richard A. Hankins, and Jignesh M. Patel. Efficient aggregation for graph summarization. SIGMOD, pages 567–580, 2008.

Tid08 Doug Tidwell. *Xslt, Second Edition*. O'Reilly Media, Inc., 2 edition, 2008.

TPAV17 Harsh Thakkar, Dharmen Punjani, Sören Auer, and Maria-Esther Vidal. Towards an integrated graph algebra for graph pattern matching with gremlin. In *Database and Expert Systems Applications - 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I*, pages 81–91, 2017.

TSK05 Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

TSL+06 Marie-Noölle Terrasse, Marinette Savonnet, Eric Leclercq, Thierry Grison, and George Becker. Do we need metamodels and ontologies for engineering platforms? In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, GaMMa '06, pages 21–28, New York, NY, USA, 2006. ACM.

vDAG12 Stijn van Dongen and Cei Abreu-Goodger. *Using MCL to Extract Clusters from Networks*, pages 281–295. Springer New York, New York, NY, 2012.

Vli02 Eric van der Vlist. *Xml Schema*. O'Reilly Media, Inc., 1 edition, 2002.

VMT15 Roberto De Virgilio, Antonio Maccioni, and Riccardo Torlone. Approximate querying of RDF graphs via path alignment. *Distributed and Parallel Databases*, 33(4):555–581, 2015.

VnI11 Adrian Viaño Iglesias. Graph representation of documents content and its suitability for text mining tasks. Master's thesis, Norwegian University of Science and Technology, Norway, 2011.

VTBL13 Elena Vasilyeva, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner. Leveraging flexible data management with graph databases. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 12:1–12:6, New York, NY, USA, 2013. ACM.

VZ14 Alejandro Vaisman and Esteban Zimányi. *Data Warehouse Systems. Design and Implementation*. Springer, 2014.

Wad00 Philip Wadler. A formal semantics of patterns in XSLT and xpath. *Markup Languages*, 2(2):183–202, 2000.

Wal07 Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., 2007.

WCH^{+}14 Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 7(9):721–732, May 2014.

Wei62 Paul M Weichsel. The kronecker product of graphs. *Proceedings of the American Mathematical Society*, 13(1):47–52, 1962.

WKS^{+}11 Stephen T. Wu, Vinod C. Kaggal, Guergana K. Savova, Hongfang Liu, Jiaping Zheng, Wendy W. Chapman, Christopher G. Chute, and Dmitriy Dligach. Generality and reuse in a common type system for clinical natural language processing. In *Proceedings of the First International Workshop on Managing Interoperability and Complexity in Health Systems*, MIXHS '11, pages 27–34, New York, NY, USA, 2011. ACM.

WM06 Christopher A. Welty and J. William Murdock. Towards knowledge acquisition from information extraction. In *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, pages 709–722, 2006.

XKS13 Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. Overlapping community detection in networks: The state-of-the-art and comparative study. *ACM Comput. Surv.*, 45(4):43:1–43:35, August 2013.

YG16 Dan Yin and Hong Gao. A flexible aggregation framwork on large-scale heterogeneous information networks. In *Journal of Information Science*, pages 1–18, February 2016.

YLW^{+}13 Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: A fast and compact system for large scale rdf data. *Proc. VLDB Endow.*, 6(7):517–528, May 2013.

ZLXH11 Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: On warehousing and olap multidimensional networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 853–864, New York, NY, USA, 2011. ACM.

ZRC+17 Ce Zhang, Christopher Ré, Michael J. Cafarella, Jaeho Shin, Feiran Wang, and Sen Wu. Deepdive: declarative knowledge base construction. *Commun. ACM*, 60(5):93–102, 2017.

ZY17 Kangfei Zhao and Jeffrey Xu Yu. All-in-one: Graph processing in rdbmss revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1165–1180, New York, NY, USA, 2017. ACM.