Technical Report

SNQL: Social Networks Query Language

TR/DCC-2011-05

Departamento de Ciencias de la Computación
Universidad de Chile

Mauro San Martín Ramas
msanmart@dcc.uchile.cl
Claudio Gutierrez
Peter T. Wood

April, 2011

Contents

Introduction

This report presents the Social Networks Query Language (SNQL) the query and transformation language of the Social Networks Data Model (SNDM).

The SNDM is a database model for the management of social networks data. This model provides a data structure to represent and store social networks, and a language to query and transform social networks (SNQL).

SNQL is inspired in data management practices from social networks analysis (SNA) practice, and from the needs that arise in the management of complex and big data from online social networks. We show that these operations can be expressed in SNQL and evaluated under reasonable complexity bounds (NLOGSPACE).

SNQL is a pattern matching and production language. Its design provides both a textual and a graphical syntax. The semantics of the language is defined in terms of GraphLog and second order tuple-generating dependencies.

This technical report is organized in two chapters. The first chapter briefly review the SNDM and describes the elements required to present its query language, in particular, the data structure. The second chapter presents the current version of the language in detail: its syntax, semantics, evaluation process and main properties (regarding expressiveness and complexity).

Chapter 1

Preliminaries: The Data Structure of the SNDM

The data structure is the component of the data model that defines how the storage requirements are resolved. The data structure must be the simplest structure so that: all objects in the domain could be represented at the proper level of abstraction, provides support to and does not hinder the performance of query language, and satisfy the additional requisites such that scalability and portability.

Accordingly, the data structure of the SNDM must be able to represent all possible social networks in terms of actors, relations, and attributes. In addition, it must support, for instance, data sharing and reuse. The requisites point to a graph-like semi-structured data model.

In this chapter, we briefly recall the main features of the data structure that are required to present the current version of query language.

1.1 Data Structure Elements

Actors, relations, and attributes are the elements whose interconnections form a social network [28].

1. Actors have a unique id and a set of attributes, and can participate in any number of relations.

2. Relations have an unique id, a set of attributes, and a number of participant actors. The number of participants can be one o more, and it may change without affecting the other properties of the relation.

3. Attributes have an associated meaning and a literal value. One attribute is identified by the id of the object to which it is attached (actor or relation), by its meaning, and by its literal value. The type (or family) of the object is a special kind of attribute (called below family).

Thus social networks are sets of actors, relations, and attributes. Sharing and reuse requires metadata at the network level to record, for instance, provenance of the data sets.

1.2 Data Structure Definition (Triples)

We present in this section the triples version of data structure. Note that depart from the classical strategy, even though the structure is still a graph, the relations are represented as nodes and the attributes are part of the network structure. We briefly introduce here a graphical syntax used to depict social networks in the following sections. We will use this notation because it is better suited to the definition and study of the query and transformation language, and also provides a better framework for implementation.

Definition 1 (Social Network Triple Representation [28]) *Consider the vocabulary* $\Sigma = A \cup T \cup C \cup F \cup \{isa, isr\} \cup L_{AT} \cup L_M$*, where* A *is the set of actor oids,* T *is the set of relations oids,* C *is the set of literal values,* F *is the set of families of actors and relations,* L_{AT} *the set of labels of participations roles of actors in relations, and* $L_M = L_{AC} \cup L_{TC}$ *is the set of label of attributes (meanings). Then define the following sets of triples:*

1. *Nodes and their Family Belonging:*
   $N \subseteq (A \times \{isa\} \times F) \cup (T \times \{isr\} \times F)$
2. *Participation roles of actors in relations:*
   $R \subseteq A \times L_{AT} \times T$
3. *Meanings (attributes):*
   $M \subseteq (A \cup T) \times L_M \times C$

From the definitions above it is not difficult to show:

Lemma 1 *A social network* $G = (V, E)$ *can be represented by three sets of triples* (N, R, M) *as described above.*

Table 1.1: Friendship network (Fig. 1.2) represented as sets of triples .

N: Typing		
a1	isa	'person'
a2	isa	'person'
a3	isa	'person'
a4	isa	'city'
a5	isa	'city'
r1	isr	'friendship'
r2	isr	'lives-in'
r3	isr	'lives-in'
r4	isr	'lives-in'

R: Roles		
a1	friend	*r1*
a2	friend	*r1*
a3	introducer	*r1*
a2	inhabitant	*r2*
a4	place	*r2*
a1	inhabitant	*r3*
a5	place	*r3*
a3	inhabitant	*r4*
a5	place	*r4*

M: Attributes		
a1	name	'Mary'
a2	name	'John'
a3	name	'Ann'
a4	name	'Capital City'
a5	name	'Central City'

Proof. It is possible to map the set of nodes and the set of edges of G to a subset of the vocabulary of the triples and to the sets of triples respectively. Each triple in can represent a labeled arc: triples in M represents edges in $E_{AC} \cup E_{TC}$ with their labels assigned by l, triples in R represents edges in E_{AT} with their labels assigned by l.

- If $A = V_A$, $T = V_T$, $C = V_C$ the set of nodes of the social network graph can be expressed as $V = A \cup T \cup C$
- Each triple in M can be expressed as (a, b, c) where $a \in A \cup T, c \in C$, and $b = l(e), e \in E_{AC} \cup E_{TC}$
- Each triple in R can be expressed as (a, b, c) where $a \in A, c \in T$, and $b = l(e), e \in E_{AT}$
- N represents the families of actors and relations. Note that the triple representation is more general, as it supports multiple families for each object.

Table 1.1 shows the triple representation of friendship network depicted in Figure 1.2 (see equivalence between graphical syntax and triples in Figure 1.1).

Graphical Syntax. Along with the data structure we define a graphical syntax to depict social networks (see Figure 1.1). It has four building blocks: actors with its family labels, relations with its family labels, participation roles of actors in relations, and attributes on actors and relations. Actors are represented by circles with their id inside, and a grey label indicating

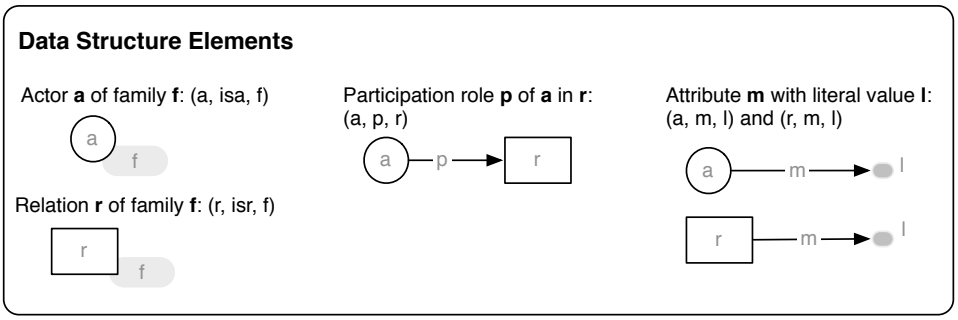

Figure 1.1: *Social Network Graphical Syntax*. Form left to right, the four graphical building blocks of a social network: an actor (above) and its family label, a relation and its family label, a participation role of an actor in a relation, and attributes on an actor and a relation. For each block it is also shown the equivalent triple representation.

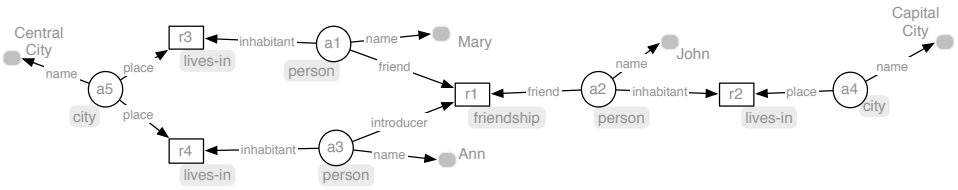

Figure 1.2: *Friendship Network*. A social network representing the friendship relation (square node) between *Mary* and *John*, who were introduced by *Ann* (actors as round nodes, and attribute values as grey dots). The cities of residence are also represented as actors.

their type. Accordingly, relations are represented by rectangles with their id inside, and a grey label indicating their type. Participation roles are represented by directed lines departing from the participating actor to the relation. Line labels differentiates participation roles.

Attributes links an actor or relation to a literal value represented by a gray dot. Line labels represent the meaning of the attribute.

Figure 1.2 depicts a small social network with two types of actors: cities and persons, both with attribute names. There are also two types of relations: lives-in linking persons and cities, friendship linking persons. For instance, it could be read "Mary lives in Central City", and "Mary and John participate as friends in r1, and Ann as 'introducer".

Chapter 2

SNQL: Query and Transformation Language

In a data model, the query language is the set of data-manipulation operations designed to access the information in a database defined under that model, that is, using the corresponding data structure. The expressiveness of the query language is the set of all queries that can be expressed in the language. A practical query language must offer an appropriate trade-off between its expressiveness and a feasible query-evaluation complexity in order to solve the actual data management problem while scaling adequately. Additionally, to actually improve users productivity, the language should be accessible and appealing to the users in the application domain.

A strategy that fulfills these conditions is pattern matching and production, whose graphical representation could be easily understood by users familiar with social networks and SNA. Thus, in a naive approach, a user could specify a basic query by drawing a pair of patterns: a motif to be searched, and a pattern to be constructed each time the motif is found. In fact, this basic strategy covers most of the basic data management needs. It is possible to show that it is equivalent to relational algebra without aggregation, and consequently to an affordable subset of Datalog [1]. However, there are requirements that are not expressible in terms of local patterns, and which require also to produce summaries and new values. To improve the expressiveness of the query language, while keeping its complexity under a practical bound (NLOGSPACE), we extended the naive approach with three elements: complex patterns (using logical combination of patterns), production of new values and object ids (using aggregate functions and second order tuple-generating dependencies), and inflationary patterns like the

subnetwork reachable from a given actor (using transitive closure). This design decision leaves out some borderline cases, like maximal and recursive cohesive subgroups (e.g. k-cores and k-plexes [32]), whose identification algorithms push the complexity over the bound stated above [7, 11]

The resulting language, SNQL, has a textual and graphical syntax, and covers all the requirements fulfilled by queries computable under the given complexity bound, which also covers all real world SN practice.

In this chapter, we present the definition of SNQL and study its properties. First, we present an overview of the language and its properties. After that, we present the theoretical background regarding Datalog and second order tuple-generating dependencies. We continue with the formal definition of the query and transformation language, and the study of its expressiveness and complexity. Finally, we briefly discuss related work on query languages for network structured data.

2.1 Query and Transformation Language Design

What is a "good" query language for the requirements given above? Although nobody has yet proposed a set of primitives, flexible and expressive enough, to represent the full diversity of queries implied by these requirements [5], the good news for SN is that the required set of functionalities seems to be small, and most of them (not all) computable by reasonable graph algorithms (paths, connectedness, etc.).

From a social networks practice point of view, we distinguish two types of operations: *data management* operations (i.e. queries and transformations) that return social networks, and *structural measures* operations that return values or sets of values for structural properties, such as centrality. SNQL concentrates on the first type, data management operations that produces networks from networks, and allows the composition of queries. As for the second group, still there are at least two ways to access structural measures at the implementation level: via import/export facilities in the DBMS from and to structural analysis tools, for instance well known SNA tools like Pajek, network and sna R packages, and UCINET; or by implementing an extension to the DBMS, a set of structural analysis functions and made them available to the query language, without the need of express them in the language itself.

2.1.1 Main Design Issues

Database queries are mappings from instances of a source schema S to instances of a target schema T. In the case of SNQL, both schemas must describe social networks.

The evaluation of a query can be seen as a two step process: first the relevant data is identified in the source instance; the resulting network is built using the identified data. It is worth noting the following issues regarding complexity and practical computational costs:

- The actual evaluation process does not need to proceed in two separate steps collecting first all relevant data in an intermediate structure and then producing the result. If a locality principle holds, it will be possible to produce a portion of the result from each selected portion of the source, thus avoiding the materialization of the intermediate data.

- Abiteboul and Vianu [2,8] show that adding value invention to Datalog makes it capable of expressing all computable queries at the expense of pushing its complexity bound over PSPACE.

- The complexity of these queries also depends on the nature of the selected and produced structures. Consens and Mendelzon [11] show that selecting and reporting whole paths, path summarizations and aggregation requires exponential time.

Consequently, to achieve a practical complexity bound, we need to narrow the set of queries in SNQL, leaving outside the language queries defined in terms of whole paths or paths summarization, such as the selection of maximal cohesive subgroups defined in terms of minimum mutual distances, e.g. k-kores and k-plexes [32]. Additionally we must consider a less powerful alternative to value invention with a feasible complexity.

The following running example introduces a friendship network along with two other networks that are defined by SNQL queries over the first. The actual queries are presented in detail later, after defining the syntax of the language.

Example 1 *Consider a SN of friendship relations among people, including some other person, if any, who introduced them; this implies having relations of variable arity (solved by representing relations as nodes). People are described by the attributes 'name' and 'city' (see Fig. 2.1(a)). To study the relevance of city of residence to friendship might require promoting cities to*

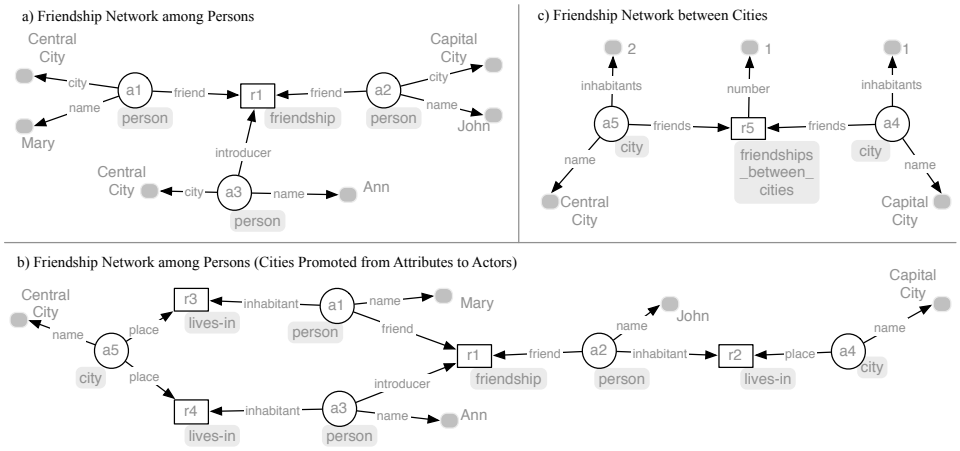

Figure 2.1: *Friendship Network and Simple Queries Results.* a) A social network representing the friendship relation (square node) between *Mary* and *John*, who were introduced by *Ann* (actors as round nodes, and attribute values as gray dots). b) The same social network after promoting *city* attributes to actors. c) The social network result after grouping persons by city and computing aggregate attributes: *inhabitants* of each city, and *number* of friendships between cities.

actors and linking people and cities with a new type of relation, e.g. 'lives-in' (see Fig. 2.1(b)). Another type of transformation would be to group people by city of residence, thus defining a network of cities, where relations summarize friendships among residents of cities. Additionally, one might like to describe in the network the population (person count) of each city, and label the relations between them with the number of friendship-relations between people (see Fig. 2.1(c)).

In the remaining of this section we present SNQL and contrast its features with the previously identified requirements and concrete examples.

2.1.2 Query Language Design

Following the requirements and general issues presented above we propose a query language composed by single query operation that collects data from the source network via pattern matching, and with this data produces the result using a second pattern as a template. Queries can be composed to form all the needed variants.

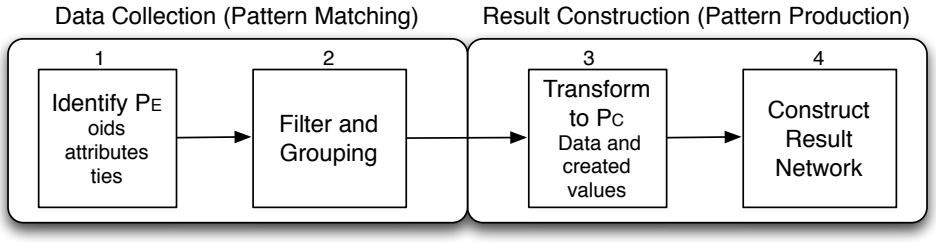

Figure 2.2: *Querying SNDM data.* The main components of a query are: an extraction pattern P_E, and a construction pattern P_C. Each query proceeds as shown in the figure: (1) P_E is matched again the source network, (2) only the subnetworks that match are extracted, and grouped if needed for aggregation, (3) values bound to elements of P_E are used to populate elements of P_C and, possibly, to create new values, (4) and finally the result network is constructed as the union of all instances of P_C.

Each query in the language is defined by two patterns elements (see Figure 2.2):

- An extraction pattern P_E is a basic SNDM network pattern, i.e. a SNDM social network (usually small) labeled with variables and constants. Constants restrict the matching of P_E, and variables are bound to the corresponding values in the source network. Each match of P_E produces a tuple of variable bindings. The basic pattern is extended with logical operations between patterns (`AND`, `OR`, `AND-NOT`), aggregating functions, and transitive closur (TC).

- A construction pattern P_C is a basic SNDM network pattern labeled with variables. The P_C is a template to produce subnetworks of the query result. Each subnetwork is produced with one tuple from the data collection process. P_C variables get their values directly from values bound to variables of P_E, or from functions that create new values or compute values using the value bound to variables from P_E

The evaluation process follows the sequence shown in Figure 2.2. First, P_E is matched against the source social network, each match produces a tuple where each variable of P_E is bound to the corresponding value found in the instance. If aggregation is specified, this collection of tuples is grouped and aggregated as needed. For each of the resulting tuples, one instance of the P_C is produced: each variable is replaced by the corresponding value,

Figure 2.3: *Graphical representation of an SNQL query.* This is the graphical representation of P_E and P_C for the query that promotes city attributes to actors producing the network depicted in Figure 2.1.b from the network depicted in Figure 2.1.a. The correspondence between variables from P_E and P_C is indicated using the same names in both patterns. $A2$ and $R2$ represent new oids created with functions: $A2 = g(L1)$ and $R2 = f(A1, A2)$.

either from a tuple variable or a predefined function, e.g.: arithmetic functions, string concatenation, value invention, etc. Finally the result is the union of all the subnetworks produced from P_E.

Example 2 *Consider the SN of friendship relations among people presented in Example 1, and the query to promote city attributes of persons to actors, graphically represented in Figure 2.3. First, P_E is matched against the friendship network: the structure and constants (attributes meanings, and family labels) must match. Each match generates a tuple of values for variables $A1$, $R1$, $P1$, $L1$, and $L2$. In turn each of these tuples is used to produce an instance defined by P_C: each value is directly used where the same variable name is indicated. New values are functionally created for new variables:$A2 = g(L1)$ and $R2 = f(A1, A2)$. (These assignments are usually expressed in the text of the query, which will be discussed in detail after the formal definition of the syntax).*

2.2 Theoretical Framework

To provide formal semantics and to study the expressiveness and complexity of SNQL, we use two known database languages: Datalog and second order tuple-generating dependencies (SO tgds).

Datalog is a query language based on formal logics that has been extensively studied, providing a good framework to characterize SNQL.

SO tgds are database formalisms to specify compositions of schema mappings. We use them to provide value invention to SNQL.

2.2.1 Datalog and GraphLog

There exist many variants of Datalog. In the presentation below we focus on the characteristics and properties useful to the study of SNQL. A specially interesting subset of Datalog is Graphlog which additionally provides a graphical syntax whose semantics is described in Datalog. Graphlog provides several features on the lines of SNQL.

Definition 2 (Datalog) *We will briefly review Datalog (for further details and proofs see [1]).*

A term *is either a variable or a constant. An* atom *is either a* predicate formula $p(x_1, ..., x_n)$*, where* p *is a predicate name and each* x_i *is a term, or an* equality formula $t_1 = t_2$ *where* t_1 *and* t_2 *are terms. A* literal *is either an atom (a* positive literal L*) or the negation of an atom (a* negative literal $\neg L$*).*

A Datalog rule *is an expression* $H \leftarrow B$ *where* H *is a positive literal called the* head[1] *of the rule and* B *is a set of literals called the* body*. A rule is* ground *if it does not have any variables. A ground rule with an empty body is called a* fact*.*

A Datalog program Π *is a finite set of Datalog rules. The set of facts occurring in* Π*, denoted* facts(Π)*, is called the* initial database *of* Π*. A predicate is* extensional *in* Π *if it occurs only in* facts(Π)*, otherwise it is called* intensional*.*

A Datalog program is non-recursive *and* safe *if it does not contain any predicate that is recursive in the program and it can only generate a finite number of answers. In what follows, we only consider non-recursive and safe programs.*

A substitution θ *is a set of assignments* $\{x_1/t_1, \ldots, x_n/t_n\}$*, where each* x_i *is a variable and each* t_i *is a term. Given a rule* r*, we denote by* $\theta(r)$ *the rule resulting from substituting the variable* x_i *for the term* t_i *in each literal of* r*.*

The meaning *of a Datalog program* Π*, denoted* facts*(Π) *, is the database resulting from adding to the initial database of* Π *as many new facts of the form* $\theta(L)$ *as possible, where* θ *is a substitution that makes a rule* r *in* Π *true and* L *is the head of* r*. Then the rules are applied repeatedly and new facts are added to the database until this iteration stabilizes, i.e., until a* fixpoint *is reached.*

[1] We may assume that all heads of rules have only variables by adding the corresponding equality formula to its body.

A Datalog query Q *is a pair* (Π, L) *where* Π *is a Datalog program and* L *is a positive (goal) literal. The* answer *to* Q *over database* $D = \text{facts}(\Pi)$ *, denoted* $\text{ans}_d(Q, D)$ *is defined as the set of substitutions* $\{\theta \mid \theta(L) \in \text{facts}^*(\Pi)\}$.

Definition 3 (Graphlog Query Graph) *A query graph [10]* G_p *is a directed labeled multigraph with a distinguished edge*

$$(N, E, L_N, L_E, \iota, \nu, \epsilon, e, L_e)$$

where: N *is a finite set of nodes,* E *is a finite set of edges,* L_N *is a set of sequences of variables,* L_E *is a set of literals and closure literals (literals followed by the positive closure operator),* ι *the incidence function,* ν *the node labeling function,* ϵ *the edge labeling function,* e *is the distinguished edge, and* L_e *is a set of positive literals; there are no isolated nodes. Edges may be labeled with literals, closure literals or path expressions, except for the distinguished edge which must be labeled with a positive non-closure literal. The distinguished edge denotes the result of the query graph when the rest of the graph matches the data source.*

Definition 4 (GraphLog) *GraphLog [10] is the query language defined by the set of graphical queries (finite sets of query graphs)* $\mathcal{G}$*, whose query graphs do not form cyclic dependencies. There is a dependence between two query graphs if one references the distinguished edge of the other. The meaning of a graphical query* $\mathcal{G}$ *can be expressed using stratified Datalog.*

GraphLog was a seminal query language for graph data, designed to be expressive while at the same time having low computational complexity. Apart from standard features, it includes aggregation and transitive closure making it suitable for many SN queries. However, GraphLog does not provide functionality to create new objects/actors, a crucial requirement for SN.

2.2.2 Second Order tgd's

Second-order tuple-generating dependencies (SO tgd's) are a type of database constraint formalism introduced to specify composition of schema mappings [15].

We use SOtgds to formalize the semantics of the construction stage of evaluation of SNQL queries. Recall that this stage requires value creation, a feature that SOtgds may provide while keeping complexity below acceptable bounds, as in the case of Datalog.

First, we define a simpler formalism which is not enough for the requirements of the language, but it is the base of SO tgds: source-to-target tuple-generating dependency.

Definition 5 (Source-to-target tgd) *Let S be a source schema and T a target schema. A source-to-target tgd is a first-order formula of the form*

$$\forall x(\phi_S(x) \rightarrow \exists y \psi_T(x, y)),$$

where $\phi_S(x)$ is a conjunction of atomic formulas over S and $\psi_T(x, y)$ is a conjunction of atomic formulas over T. Note that x, y are tuples/sets of variables. We assume that every variable in x appears in ϕ_S.

Definition 6 (Full source-to-target tgd) *A full source-to-target tgd is a source-to-target tgd of the form*

$$\forall x(\phi_S(x) \rightarrow \psi_T(x)),$$

where $\phi_S(x)$ is a conjunction of atomic formulas over S and $\psi_T(x, y)$ is a conjunction of atomic formulas over T. We again assume that every variable in x apperas in ϕ_S.

Source-to-target tgds have been used to formalize data exchange. Moreover, they have been used in data integration scenarios under the name of GLAV assertions.

Example 3 *Consider the following three schemas S_1, S_2 and S_3. Schema S_1 consists of a single binary relation symbol $Takes$, that associates student names with the courses they take. Schema S_2 consists of a similar binary relation symbol $Takes1$ and of an additional binary relation symbol $Student$ that associates each student name with a student id. Schema S_3 consists of one binary relation symbol $Enrollment$ that associates student ids with the courses the students take. Consider now the schema mappings $M_{12} = (S_1, S_2, \Sigma_{12})$ and $M_{12} = (S_2, S_3, \Sigma_{23})$, where*

$$\Sigma_{12} = \{\forall n \forall c(Takes(n, c) \rightarrow Takes_1(n, c)),$$

$$\forall n \forall c(Takes(n, c) \rightarrow \exists s Student(n, s))\}$$

$$\Sigma_{23} = \{\forall n \forall s \forall c(Student(n, s) \wedge Takes_1(n, c) \rightarrow Enrollment(s, c))\}$$

These three formulas are source-to-target tgds. The second formula in Σ_{12} is an example of a source-to-target tgd that is not full, while the other two

formulas are full source-to- target tgds. The first mapping, associated with the set Σ_{12} of formulas, requires that copies of the tuples in Takes must exist in Takes1 and, moreover, that each student name must be associated with some student id (s) in Student. The second mapping, associated with the formula in Σ_{23}, requires that pairs of student id and course must exist in the relation Enrollment, provided that they are associated with the same student name.

SNQL requires the functional creation of values; *source-to-target tgds* are not enough. A slight extension with existential second-order features, *Second-Order tgds*, which are *source-to-target tgds* extended with existentially quantified functions and equalities. A particular case of SO tgds with predefined functions provides the required functionality.

Definition 7 (SO tgd) *Let* $\mathbf{x}$ *be a collection of variables and* $\mathbf{f}$ *be a collection of function symbols. A* term *is defined a follows: every variable in* $\mathbf{x}$ *is a term, and if* f *is a* k*-ary function symbol in* $\mathbf{f}$ *and* $t_1, \ldots, t_k$ *are terms, then* $f(t_1, \ldots, t_k)$ *is a term. Let* $\mathbf{S}$ *be a source schema and* $\mathbf{T}$ *a target schema. A* second order tuple-generating dependency *(SO tgd) is a formula of the form:*

$$\exists \mathbf{f}((\forall \mathbf{x_1}(\phi_1 \rightarrow \psi_1)) \wedge \ldots \wedge (\forall \mathbf{x_n}(\phi_n \rightarrow \psi_n)))$$

1. *Each member of* $\mathbf{f}$ *is a function symbol.*
2. *Each* ϕ_i *is a conjunction of*
   - *atomic formulas of the form* $R(y_1, \ldots, y_k)$*, where* R *is a* k*-ary relation symbol of schema* $\mathbf{S}$ *and* $y_1, \ldots, y_k$ *are variables in* $\mathbf{x_i}$*, not necessarily distinct,*
     and
   - *equalities of the form* $t = t'$ *where* t *and* t' *are terms based on* $\mathbf{x_i}$ *and* $\mathbf{f}$*.*
3. *Each* ψ_i *is a conjunction of atomic formulas* $S(t_1, \ldots, t_l)$ *where* S *is an* l*-ary relation symbol of schema* $\mathbf{T}$ *and* $t_1, \ldots, t_l$ *are terms based on* $\mathbf{x_i}$ *and* $\mathbf{f}$*.*
4. *Each variable in* $\mathbf{x_i}$ *is a safe term with respect to* ϕ_i *and* $\mathbf{f}$*. A* safe term with respect to ϕ_i and $\mathbf{f}$ *is defined recursively as one of the following: (a) a variable* x *occurring in a relational atomic formula of* ϕ_i*, (b) a*

variable x occurring in an equality term of the form $x = t$ or $t = x$ of ϕ_i, where t is a safe term with respect to ϕ_i and $\mathbf{f}$, or (c) a term $f(t_1, \ldots, t_k)$ where f is in $\mathbf{f}$ and $t_1, \ldots, t_k$ are safe terms with respect to ϕ_i and $\mathbf{f}$.

The fourth condition is a"safety" assumption –not to be confused with the safe condition in Datalog– that makes the second-order tgds domain independent (so that their truth does not depend on any underlying domain, but only on the "active domain" of elements that appear in tuples in the source). In the case of first-order tgds, where equalities are not present, this condition becomes a simpler one by requiring that every universally quantified variable appear in one of the relational atomic formulas in the left-hand side of the tgd. This condition is not always made explicit in the literature in the definition of first-order tgds, although it should be.

The following formula is a valid example of a second-order tgd where all universally quantified variables are safe:

$$\exists f \forall x \forall y \forall z (R(x) \wedge (y = f(z)) \wedge (z = x) \rightarrow S(x, y, z))$$

Example 4 *Consider the following three schemas S_1, S_2 and S_3. Schema S_1 consists of a single unary relation symbol Emp of employees. Schema S_2 consists of one binary relation symbol Mgr1, that associates each employee with a manager. Schema S_3 consists of a similar binary relation symbol Mgr and an additional unary relation symbol SelfMgr, intended to store employees who are their own managers. Consider now the schema mappings $M_{12} = (S_1, S_2, \Sigma_{12})$ and $M_{23} = (S_2, S_3, \Sigma_{23})$, where*

$$\Sigma_{12} = \{\forall e(Emp(e) \rightarrow \exists m Mgr_1(e, m))\}$$

$$\Sigma_{12} = \{\forall e \forall m(Mgr_1(e, m) \rightarrow Mgr(e, m)),$$
$$\{\forall e(Mgr_1(e, e) \rightarrow SelfMgr(e))\}$$

It is easy to verify that the composition of M_{12} and M_{23} is M_{13}, where Σ_{13} is the following SO tgd:

$$\exists f(\forall e(Emp(e) \rightarrow Mgr(e, f(e))) \wedge \forall e(Emp(e) \wedge (e = f(e)) \rightarrow SelfMgr(e)))$$

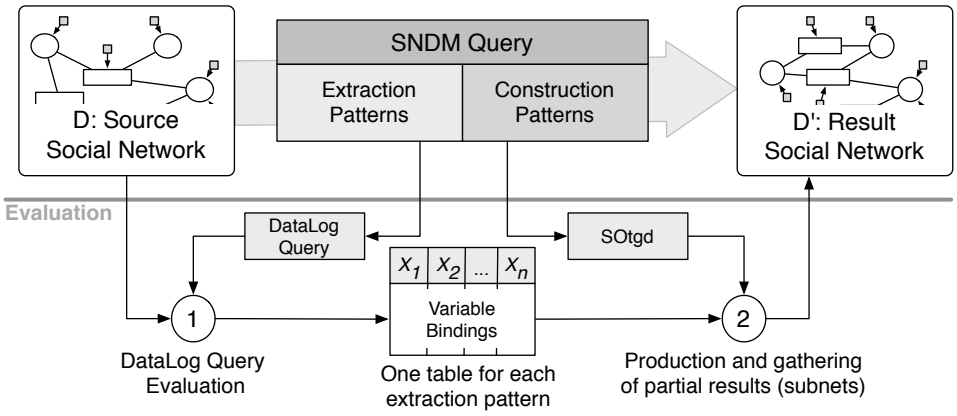

Figure 2.4: *The evaluation process of a SNQL query.* Evaluation process occurs in two stages: (1) a Datalog program equivalent to `<extract-patt>` is evaluated over the data source network D producing intermediate tables of variable bindings; (2) partial results are produced from the obtained tuples (a table for each distinguished predicate) using the SO tgd equivalent to `<construct-patt>`. All partial results are gathered in the final result network D'.

2.3 Syntax and Semantics

SNQL is inspired by two earlier languages: GraphLog [10] and second-order tuple-generating dependencies (SO tgds) [15]. Its evaluation process is syntactically and conceptually built in two modules (see Figure 2.4), SN matching and SN construction, which essentially are GraphLog and SO tgds respectively.

We formally define SNQL specifying its syntax and semantics. SNQL syntax has two versions: a text syntax that follows the lines of SQL and other similar languages, and a graphical syntax for patterns that extends the graphical syntax of SNDM instances to represent variables and complex extraction patterns. We define the semantics of SNQL following its two stages. The collection of relevant data from the source networks is defined in terms of Datalog, and the building of the resulting network is defined using SOtgds. This formal framework allows to study the complexity and expressiveness of the SNQL.

2.3.1 Query and Transformation Language: Syntax

The transformation and query language should be friendly enough for both the lay-user and the programmer. For the first, a visual language close to the SN graphical representation is ideal: in the simpler cases, one extraction pattern and one construction pattern cover many use cases; in the general case, the extraction pattern should resemble the DAG of query graphs that exists in GraphLog [10]. For the programer, an SQL-like language would be familiar to developers and advanced database users (for searching text, writing, pasting, debugging, etc.) Thus, our language has both syntaxes.

At the abstract level, and for the purpose of formally studying and analyzing its semantics and complexity, we use a translation to a more formal representation, based on Datalog/GraphLog [1, 10] and SOtgds [15].

Definition 8 (SNQL Query) *An SNQL query Q follows the standard SELECT|CONSTRUCT – WHERE – FROM structure of languages like SQL and SPARQL. It receives social networks as input (the FROM clause), extracts information using patterns (the WHERE clause), and outputs a new social network, possibly with new values, using the CONSTRUCT clause.*

```
Q ::= CONSTRUCT <list-of-construct-patt>
      WHERE     <extract-patt>
      FROM      <list-of-social-networks>
```

Each construction pattern *in* `<list-of-construct-patt>` *is a collection of* `<list-of-patt-triples>` *(set of triples including variables) possibly constrained by a list of equalities:*

```
<construct-patt> ::= <list-of-patt-triples>
  [IF <expr> = <expr>[ AND <expr> = <expr>]*] [AS <sn-id>]
```

The `<extract-patt>` *is either a* basic pattern `<list-of-patt-triples>` *(to be matched against one of the* social networks *listed in the* `FROM` *clause), or a complex pattern built using operations between patterns:*

```
<extract-patt>  ::=  <list-of-patt-triples> [MATCH <sn-id>]
 | <extract-patt> AND <extract-patt>
 | <extract-patt> OR <extract-patt>
 | <extract-patt> AND-NOT <extract-patt>
 | <extract-patt> FILTER <condition>
 | TC(<startV>, <endV>, <extract-patt>) WITH <start-condition>
 | AGG(<group-vars>, <aggr-func>, <extract-patt>)
```

where `TC` *denotes transitive closure and* `AGG` *denotes aggregation, both explained further below. Finally* `<list-of-social-networks>` *is a list of sources (social networks) with aliases.*

```
<list-of-social-networks> ::= <social-network> [AS <sn-id>
                              [, <social-network> AS <sn-id>]*]
```

Let's recall that a social network is a collection of triples composed from object identifiers and constant literals. See the complete syntax in Figure 2.5.

Example 5 *Consider again the SN from Example 1. The following SNQL query produces the network depicted in Fig. 2.1(b) from that in Fig. 2.1(a) by promoting the 'city' attribute to a new type of actor (*city*) and producing a new type of relation (*lives-in*) to associate people with cities.*

```
CONSTRUCT CP  IF R2 = f(A1, A2) AND A2 = g(L1)
  WHERE   EP
  FROM    FriendshipNetwork
```

Patterns `EP` *and* `CP`*, depicted in Fig. 2.3, denote an* extraction pattern *and a* construction pattern*, respectively.* `FriendshipNetwork` *is the SN shown in Fig. 2.1(a).*

Note that it is also possible to represent both patterns directly in the text:

```
CONSTRUCT {(A1, isa, person), (A2, isa, city), (R1, isr, friendship),
           (R2, isr, lives-in), (A1, inhabitant, R2), (A1, P1, R1),
           (A1, name, L2), (A2, place, R2), (A2, name, L1)}
          IF R2=f(A1, A2) AND A2=g(L1)
  WHERE   {(A1, isa, person), (R1, isr, friendship),
           (A1, city, L1), (A1, P1, R1), (A1, name, L2)}
  FROM    FriendshipNetwork
```

Note that in the result, cities become hubs that connect all people living in each of them, and that the new actors require creation of new ids from the data: the oids of cities are produced by applying a function g to the literal values bound to variable `L1`*. Similarly, new relation identifiers for the 'lives-in' relation are created, one for each (person,city) pair matched by* `(A1,A2)`*.*

The following should be noted with regard to the syntax:

- `<condition>` is a Boolean expression over pattern variables and constants.

```
<SN-expression>            ::= <SN-query> | <social-network>
<SN-query>                 ::= CONSTRUCT <list-of-construct-patt>
                                  WHERE  <extract-patt>
                                  FROM   <list-of-social-networks>
<list-of-construct-patt>   ::= <construct-patt>[ AS <sn-id>
                                 [, <construct-patt> AS <sn-id>]*]
<construct-patt>           ::= <list-of-pattern-triples>
                                           [IF <list-of-equalities>]
<list-of-equalities>       ::= <expr> = <expr>
                                           [AND <expr> = <expr>]*
<extract-patt>             ::= <list-of-pattern-triples>
                                           [MATCH <sn-id>]
                             | <extract-patt> AND <extract-patt>
                             | <extract-patt> OR <extract-patt>
                             | <extract-patt> AND-NOT
                                                  <extract-patt>
                             | <extract-patt> FILTER <condition>
                             | TC(<startV>,<endV>,<extract-patt>)
                                              WITH <start-condition>
                             | AGG(<group-vars>,<aggr-func>,
                                                 <extract-patt>)
<list-of-pattern-triples>  ::= {<pattern-triple>
                                            [, (<pattern-triple>)]*}
<pattern-triple>           ::= (term, term, term)
<list-of-social-networks>  ::= <social-network>[ AS <sn-id>
                                 [, <social-network> AS <sn-id>]*]
<social-network>           ::= <sn-id>
                             | <list-of-instance-triples>
<list-of-instance-triples> ::= {<instance-triple>
                                           [, (<instance-triple>)]*}
<instance-triple>          ::= <n-triple>
                             | <r-triple>
                             | <m-triple>
<n-triple>                 ::= (<oid>, <constant>, <constant>)
<p-triple>                 ::= (<oid>, <constant>, <oid>)
<m-triple>                 ::= (<oid>, <constant>, <constant>)
<constant>                 ::= <object-id> | <literal>
<condition>                ::= <logic-expression>
<term>                     ::= <variable> | <constant>
                             | <expr> AS <alias>
<expr>                     ::= <variable> | <constant>
                                  | <function>(<expr>[, <expr> ]*)
<alias>                    ::= <variable> | <null>
```

Figure 2.5: SNQL Syntax.

- For `TC` the variables `<startV>` and `<endV>` must appear in `<extract-patt>`. Then `TC` returns the transitive closure of the binary relation formed by all instantiations of `<startV>` and `<endV>` when matching `<extract-patt>`. `TC` allows the specification of an additional condition `<start-condition>` that is applied only to be able to initially locate the first match of the recurrent pattern.

- `AGG` returns a tuple of values comprising each distinct instantiation of the variables in `<group-vars>` along with the result of applying `<aggr-func>` to the remaining variables in `<extract-patt>`.

Much like in SQL, the language allows to work with multiple source social networks which, in turn, may allow to express some set theoretical operations with these queries (see Section **??**).

It is worth mentioning a straightforward extension of SNQL: to produce tables instead of networks. In this case, each instance of the construction pattern in the output could be treated as a tuple; producing a table instead of a network as a result (as in SPARQL SELECT queries). This extension is not pursued in this work given the emphasis in to allow the composition of queries.

Extending the graphical syntax for network instances, allowing variables as labels and introducing elements to represent complex patterns, it is possible to graphically represent both extraction and construction patterns (see Figure 2.6).

2.3.2 Query and Transformation Language: Semantics

An SNQL query maps social networks into social networks. At the abstract level, the semantics of the SNQL query `(CONSTRUCT <T> WHERE <PATT> FROM <S>)` can be considered as comprising two steps: (1) generate an intermediate table $Temp$ formed from the results of matching of pattern `PATT` against the network `S`, (2) construct a network, using the triples in `T` instantiated with the values in the table $Temp$.

Let D be a social network, Q an SNQL query, and $Q(D)$ the result of applying Q to D. For a set of variables X, let $\overline{x}$ be the tuple comprising all variables in X.

Extraction Semantics An extraction pattern is recursively decomposed and simulated by a Datalog program as follows: let `PATT` be the pattern to be simulated by predicate p and assume that patterns `PATT1` and `PATT2`

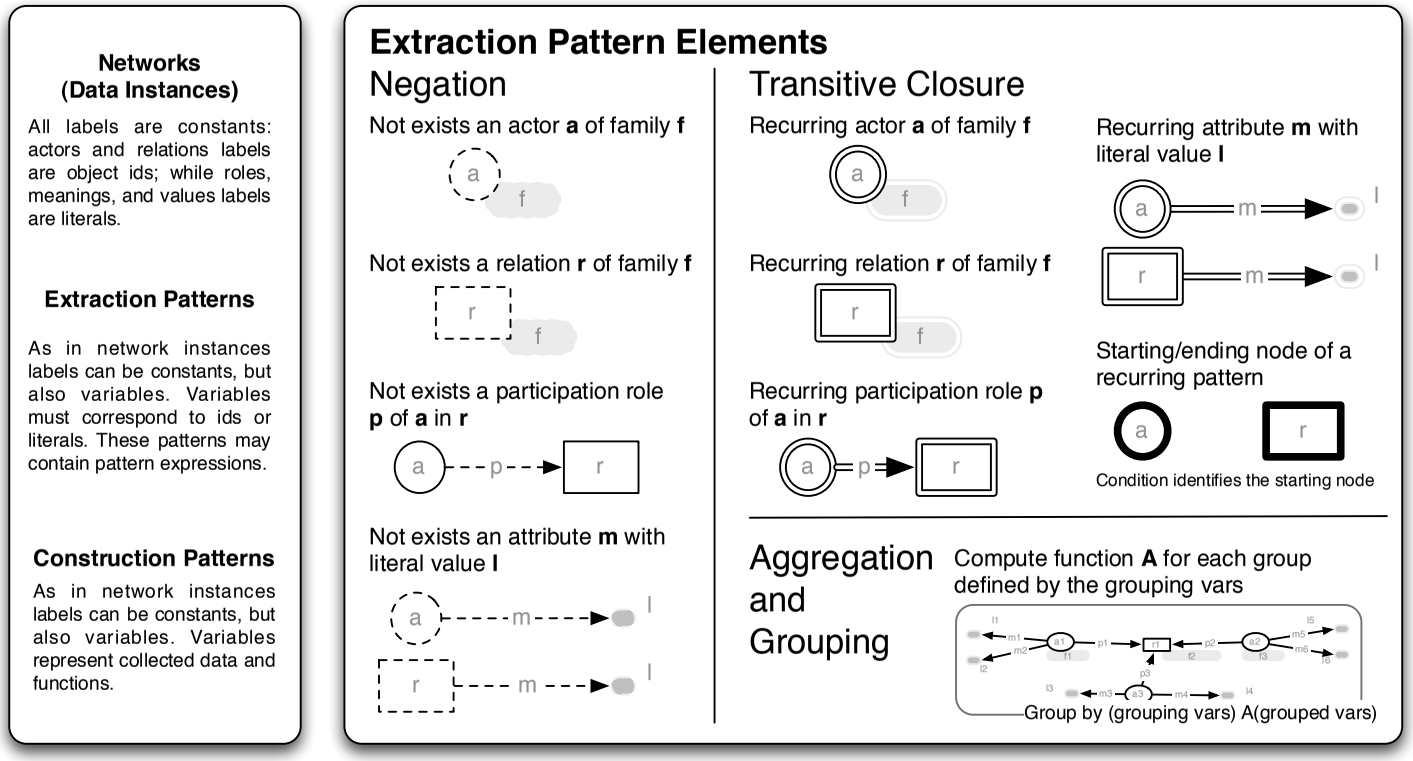

Figure 2.6: *Graphical syntax for patterns.*

0. Each triple `t` of the form `(A,B,C)` is translated as $t(A, B, C)$.

1. A list of triples (basic pattern) `{ t1, ..., tn }`:
$$p(\overline{z}) \leftarrow \bigwedge_{i \in 1..n} t_i(A_i, B_i, C_i).$$

2. `PATT1 AND PATT2`: $p(\overline{z}) \leftarrow p_1(\overline{x}), p_2(\overline{y})$

3. `PATT1 OR PATT2`: $p(\overline{z}) \leftarrow p_1(\overline{x})$
$$p(\overline{z}) \leftarrow p_2(\overline{y})$$

4. `PATT1 AND-NOT PATT2`: $p(\overline{z}) \leftarrow p_1(\overline{x}), \neg p_2(\overline{y})$.
(We cannot enforce safe negation, hence we rely on closed world assumption to ensure tractability.)

5. `PATT1 FILTER C`: $p(\overline{z}) \leftarrow p_1(\overline{x}), c(\overline{x})$
(assuming condition `C` is simulated by predicate c)

6. `TC (Vs, Vt, PATT1) WITH <start-condition>`:
$$p(U, V) \leftarrow p_1(\ldots U \ldots V \ldots), start\_cond(\ldots U \ldots V \ldots)$$
$$p(U, V) \leftarrow p_1(\ldots U \ldots W \ldots), p(W, V)$$
(assuming variable `Vs` corresponds to variable U and `Vt` to variable V of $p_1(\overline{x})$)

7. `AGG(VList, AggF, PATT1)` : $p(\overline{z}, \mathbb{A}(\overline{y})) \leftarrow p_1(\overline{z}, \overline{y})$
(assuming `Vlist` is the set of variables Z, $Y = X - Z$ and `AggF` is the aggregate function $\mathbb{A}$)

Figure 2.7: Translation of Extraction Pattern to Datalog.

are simulated by p_1 and p_2, respectively. Let $\overline{z}$, $\overline{x}$, and $\overline{y}$ contain the projected variables of `PATT`, `PATT1` and `PATT2`, respectively. Depending on the structure of `PATT`, the translation is as shown in Figure 2.7.

Construction Semantics The predicate p, obtained from pattern `PATT` in the previous translation, is now used to produce the query result. Here the list of triples `<list-of-pattern-triples>` of the `CONSTRUCT` clause along with the corresponding lists of equalities `<expr> = <expr>` play a central role. The equalities are of two types. One type defines each variable: $v_i = term_i$, $1 \leq i \leq k$; the other is of the form $term_i = term_l$, where each term may contain variables (from p), constants and functions.

For a given `CONSTRUCT trList IF eqList`, the construction process takes the result of the extraction process, the $p(\overline{z})$ predicate, plus the list of equalities `eqList` translated as $\wedge_j eq_j$ to produce the following rule:

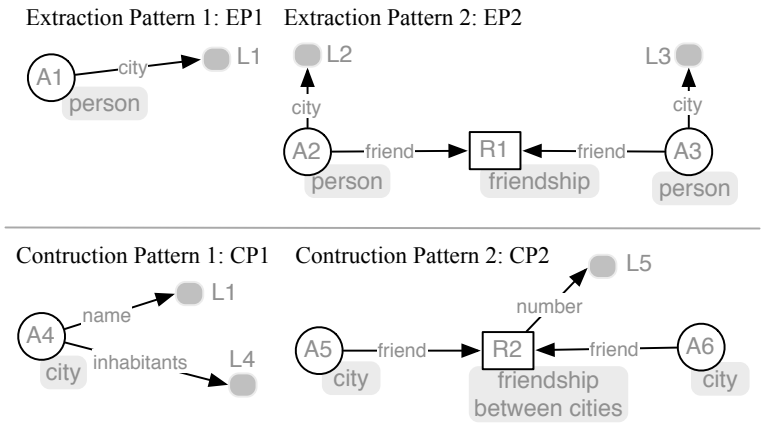

Figure 2.8: *Grouping and Aggregation.* Patterns `EP1` and `CP1` group people by 'city' and count the number of *inhabitants* in each city. Patterns `EP2` and `CP2` group and count friendship relations between pairs of cities.

$$construct(v_1, \ldots, v_k) \leftarrow p(\overline{z}) \wedge \bigwedge_j eq_j. \tag{2.1}$$

Finally, the resulting social network SN is the set of instantiations of each triple t in the list of triples in the `CONSTRUCT` using the values in the *construct* predicate:

$$\begin{aligned} SN \quad = \quad \bigcup \Big\{ & t(u_1, u_2, u_3) : \exists(..u_1..u_2..u_3..) \in construct, \\ & \text{and } t \text{ in } \texttt{trList} \Big\}. \end{aligned} \tag{2.2}$$

In practice, the evaluation process may proceed more efficiently by avoiding intermediate materialization: each match of the extraction pattern produces a collection of tuples corresponding to Datalog predicates, and this collection of tuples is processed by the construction pattern to produce a subnetwork of the output.

Example 6 *(Grouping and Aggregation) The following SNQL query Q produces the network depicted in Fig. 2.1(c) from that in Fig. 2.1(a), grouping people by city and counting friendship relations between cities.*

```
CONSTRUCT CP1 IF A4 = f(L1) AS SN1
  WHERE   AGG({L1}, COUNT AS L4, EP1)
```

```
output-n(A4,isa,city)       :- construct1(A4,L1,L4)
output-m(A4,name,L1)        :- construct1(A4,L1,L4)
output-m(A4,inhabitants,L4) :- construct1(A4,L1,L4)

output-n(A5,isa,city)       :- construct2(A5,R2,A6,L5)
output-n(R2,isr,
 friendship-between-cities) :- construct2(A5,R2,A6,L5)
output-n(A6,isa,city)       :- construct2(A5,R2,A6,L5)
output-r(A5 friend,R2)      :- construct2(A5,R2,A6,L5)
output-r(A6,friend,R2)      :- construct2(A5,R2,A6,L5)
output-m(R2,number,L5)      :- construct2(A5,R2,A6,L5)

construct1(A4,L1,L4)        :- ag1(L1,N), A4=f(L1), L4=N
ag1(L1,count(A1))           :- ep1(A1,L1)
ep1(A1,L1)                  :- n(A1,isa,person), m(A1,city,L1)

construct2(A5,R2,A6,L5)     :- ag2(L2,L3,M), A5=f(L2), A6=f(L3),
                               R2=g(A5,A6), L5=M
ag2(L2,L3,count(A2,R1,A3))  :- ep2(A2,A3,R1,L2,L3)
ep2(A2,A3,R1,L2,L3)         :- n(A2,isa, person),
                               n(R1,isr,friendship),
                               n(A3,isa,person), r(A2,friend,R1),
                               r(A3,friend,R1), m(A2,city,L2),
                               m(A3,city,L3), L2 != L3
```

Figure 2.9: Translation of query in Example 6 to Datalog.

```
  FROM    FriendshipNetwork
UNION
CONSTRUCT CP2 IF A5 = f(L2) AND A6 = f(L3) AND R2 = g(A5, A6) AS SN2
  WHERE   AGG({L2,L3}, COUNT AS L5, EP2 FILTER (L2 != L3))
  FROM    FriendshipNetwork
```

Patterns EP1, EP2, CP1, *and* CP2 *are depicted in Fig. 2.8. Note that each new group (actor) requires a new oid functionally produced from the value of attribute 'city'. Also the number of* inhabitants *bound to* L4 *and the number of* friendship-between-cities *bound to* L5 *must be computed with the aggregate function* COUNT*. The first argument of* AGG *is the set of grouping variables; the second is the aggregation function required; and the third is an extraction pattern. The results of the two construct queries are combined using* UNION *to produce the desired result.*

The translation of Q to Datalog is shown in Figure 2.9.

2.4 Expressiveness and Complexity

2.4.1 Expressiveness

SNQL is composed of two modules: one for extracting of information and another for constructing a new network. In the design, consideration has been given to providing the maximum expressiveness possible, while keeping the complexity of processing within reasonable bounds. First, for extraction stage, we considered GraphLog (possibly with summarization functions), which is a graph query language designed to be simple, graphical, oriented to graphs, and be as expressive as possible while staying within the LOGSPACE complexity bound [10]. Second, for the construction module, whose main purpose is the creation of new identifiers in the process of creating the new network, the language is modeled after second-order tuple-generating dependencies, which are known to be a family of transformations between tables of tuples with the "right" expressiveness/complexity tradeoff [15].

Knowing this, it comes as no surprise that SNQL covers all use cases we identified as being used in current practice by SN researchers. (There are still some queries defined theoretically by SN scientists which are not covered by SNQL; but, it can be proved that they fall out of the scope of a reasonably efficient complexity bound. A typical example is the cohesive subgroups defined in terms shortest paths lengths between members, for instance *k-cores*.)

Formally stated, this result can be presented as follows:

Claim: *SNQL solves all use cases presented in SN practice that fall in the NLOGSPACE complexity bound.*

A formal proof of this claim relies on the list of use cases in current practice. The column "Required Query Features" of Table **??** collects the features needed for the classical use cases from the SN community. All of them, except induced subgraph, are incorporated directly in the language.

As for the expressive power as compared to classical databases languages, we can prove the following two results:

Theorem 1 *The SNQL extraction module has the same expressive power as GraphLog.*

Proof. From the semantics of the translation to Datalog presented (see Figure 2.7), it is clear that every `<extract-patt>` can be expressed in GraphLog.

The interesting part to show is that all GraphLog *graphical queries* can be simulated by an `<extract-patt>`. The semantics of a GraphLog *query graph* are also defined by translation to Datalog, producing rules very similar to those in Figure 2.7. Hence, for such rules, it is not difficult to see that they can be expressed as patterns by considering the translation given in Figure 2.7 in the reverse direction. A GraphLog *graphical query* is a set of query graphs whose dependence graph is acyclic. For simulating a GraphLog graphical query, the intuitive idea is: `<extract-patt>` can simulate all dependency trees of patterns by un-threading the acyclic graph and transforming it into a tree.

Theorem 2 *The construction module can be specified by one SOtgd of the form:*

$\exists f_1 \ldots f_m(\forall \overline{x_1}(\phi_1 \rightarrow \psi_1) \wedge \ldots \wedge \forall \overline{x_n}(\phi_n \rightarrow \psi_n))$,
where each $(\phi_i \rightarrow \psi_i)$ *has the form:*

$$(p(\overline{x}) \wedge \bigwedge_k eq_k) \rightarrow (t_1 \wedge \ldots \wedge t_r), \tag{2.3}$$

where $p(\overline{x})$ *and* eq_k *follow the notations of equations (2.1) and (2.2), that is, predicate* $p(\overline{x})$ *is the result of the processing of the extraction pattern, and the* t_j *and* eq_k *are predicates resulting from the translation of the triples and equations in the CONSTRUCT clause, and each tuple* $\overline{x_i}$ *includes all variables in* p *and in the* eq_k*'s.*

Proof. Note that from their semantics it follows that the evaluation of each expression `<list-of-triples> IF (<exp> = <exp>)*` over the results of `<extract-patt>`, can be simulated by a formula of the form $(\phi_i \rightarrow \psi_i)$ (see eq. (2.3)) where $p(\overline{x})$ corresponds to the translation of the pattern as in the previous theorem. Hence the conjunction of these formulas can simulate the whole `CONSTRUCT` expression. This is precisely a SO tgd. (Note that SNQL uses predefined functions in the CONSTRUCT clause, while SO tgds allow existential functions.)

2.4.2 Complexity

A naive implementation of the semantics presented in Figure 2.7 (see also Fig. 2.4) would materialize intermediate results. This can be avoided by using the algorithm in Figure 2.10.

Lemma 2 *The Evaluation Algorithm is correct—it preserves SNQL semantics.*

(a) For an expression

```
CONSTRUCT trList1 IF eqList1;
          ...
          trListn IF eqListn;
```

translated in the form of eq. (2.3), define n clauses:
$aux_j(\overline{x_j}) \leftarrow p(\overline{x}) \wedge \bigwedge_k eq_k, \quad j = 1, \ldots, n.$

(b) For each clause `trListj IF eqListj;` and each triple `(x,y,z)` in `trListj`, define a rule
$t(x, y, z) \leftarrow aux_j(\overline{x_j})$.

(c) Add the clauses generated by (a) and (b) to the original GraphLog Program generated from the extract module.

(d) Obtain the values of the triples to be generated by running the new program.

Figure 2.10: Evaluation Algorithm.

Proof. First, note that the resulting program is a Datalog program. (The clauses added are simply predicates plus equalities.) Hence, it follows that this evaluation procedure gives the correct result (given by the original semantics). This is only possible, though, because of a subtle point: each clause $\phi_j \rightarrow \psi_j$ in the SO tgd can be evaluated independently because we have fixed functions. (If not, the rules would not be independent. For example, assume $a = f(x)$ is in `eqList1` and $b = f(x)$ in `eqList2` for a given variable function f. Then, if a given value of x fires rule 1, then it should not fire rule 2.)

We will show that, from a database perspective, the above evaluation computes queries efficiently. As is customary when studying the complexity of the evaluation problem for a query language [31], we consider its associated decision problem. We denote this problem by EVALUATION and define it as follows.

INPUT : A Social Network S, a query Q and a triple $t = (a, b, c)$.
QUESTION : Is $t \in [[Q]]_D$?

Theorem 3 *The complexity of* EVALUATION *is in NLOGSPACE.*

Proof. Note that adding the clauses in (a) and (b) of the Evaluation Algorithm to the GraphLog program L produces a Datalog program, and the size of that program stays in the bounds of the original query. Hence to check if a given triple t is in $[[Q]]_D$, we just have to instantiate each t in (b), hence in aux_j. Then, check the recursive equations in (a). Now check if $p(\overline{x})$ is in the GraphLog program left (note that it has no functional equations now). It is known that GraphLog can be evaluated in NLOGSPACE [10].

2.5 Related Work

The three most similar proposals, that we are aware of, oriented to social networks and with a related aim and scope are: BiQL [13], SocialScope [4], and SoQL [27].

BiQL is an SQL-like language design with a set of features that shares the motivation of providing database support for SN and interoperation with analysis tools; however the authors do not provide an implementation, nor a study of its complexity properties. From our point of view, a major drawback is the choice of the data structure. Although, at a low level the underlying graph is represented with a set of tables similar to the ones used in SNDM; these low level structures are exposed through the query language to the users, which is avoided by design in SNDM. It is worth mentioning that this model allows the creation of links among links –in this proposal there is no distinction between nodes and edges–, which is intentionally forbidden in SNDM. BiQL query language works selecting data using path expressions, and building collections of nodes and links, called *domains* –one of the low level structures we refer above. SNQL patterns are more general structures, consequently, SNQL can express all the BiQL examples and cases discussed by the authors. Given the stated similitudes, it is a reasonable assumption that BiQL complexity should be similar or lower than the complexity of SNQL.

SoQL is also an SQL-like query language design for SN; it is focused on identifying paths and groups. In this model a SN is composed by actors and binary relations. SoQL defines three types of queries SELECT FROM PATH, SELECT FROM GROUP, and CONNECT USING PATH/GROUP. In each case the key feature is the ability to find paths and groups that fulfill given predicates. The syntax support multiple aggregation levels and returning whole paths and summaries, which does not even ensure the practical tractability of computing complete answers of queries [11]. Although the authors seem aware of the complexity problems posed by their design, they

do not provide any complexity assessment, they only suggest the use of deployment parameters to ensure the practicability of SoQL query evaluation (e.g. the maximal time to be spent on a single query evaluation).

SocialScope is a logical architecture for discovering, integrating, and managing social information in social content sites; thus, its scope goes beyond data management for this particular context. It comprises three layers: content management, information discovery, and information presentation. SocialScope framework includes an algebraic language for manipulation of social content graphs (which are vey large social networks that includes a rich variety of objects); its main objective is to enable uniform data manipulation of social content graphs. The algebraic language includes selection and set theoretical operations, along with binary operations to combine networks and aggregation.

We think that, from the three proposals discussed here, SocialScope is the closely related to SNDM/SNQL. SocialScope is strongly founded on a practical problem (examples come from Y!Travel service), and provides – among other features– a query language for its data managing requirements. We produced a translation of SocialScope query language to SNQL, and we found that SNQL covers the query language of SocialScope.

There exists data management support in several fields which deal with networks [9, 16, 20], however to the best of our knowledge there is no systematization of data models for social networks. Jensen and Neville [21] propose Proximity, a data mining tool that supports statistical models over network data, which in turn uses MonetDB, a DBMS able to store networks, but lacking the abstraction level for SNA use. Proximity uses a graph query language called QGraph [6]. In the context of SNA, QGraph suffers from two major drawbacks: the results of the queries are collections of subgraphs (not a new consolidated network), and it models relations as links. Furthermore, as far as we know there is no implementation of network transformations in Proximity. Among the works which are more related to our proposal is that of Güting [17], which concentrates on query capabilities, as opposed to ours which also considers transformation issues. To get a broader view of the developments in this area, the reader can consult a survey on graph databases [5]. In [28] the authors show how to use RDF and SPARQL to query and transform SN in the semantic web. That work presents a three layered language based on SPARQL. In the present work we define an abstract language, with enhanced flexibility and expressive power (including creation of values and transitive closure), using developments and results of classical databases like GraphLog [10, 11] and second order tuple-generating dependencies [15].

There are several SNA software tools. A good example is Pajek [12], which offers an environment for exploratory graphical SNA along with a complete set of analysis algorithms. It includes basic elements of data manipulation, for instance to filter nodes and edges based on attributes. However, as with other SNA tools, data management is limited by a storage system based on text files containing edge lists or adjacency matrices. Additionally, low-level representation restrictions are exposed to the user complicating data manipulation unnecessarily. For instance, Pajek requires that in every network actor identifiers be consecutive integers starting at 1, which transfers to the user the burden of keeping track of actor identity when working with imported data, and when, for instance, a series of data manipulation operations eliminates some actors and changes the identifiers of all remaining actors.

The *R* statistical software also deals with network data [19]. Here too, data files are used to provide persistence for main memory data structures, not to support data management operations. The aim of the *Network Workbench* project[2] is to build and support the infrastructure to provide access to a repository of data and algorithms related to large-scale network analysis. They plan to provide tools to facilitate interoperation, but they do not contemplate the definition of a specialized data model. There are many custom-made applications that solve specific problems, for instance Klink et al. [24] use explicit social network data to improve DBLP (Digital Bibliography and Library Project) navigation experience by including social network structures in a search interface. Tsvetovat et al. [29] propose a specific application based on the *relational data model*, and also propose DyNetML [30], an XML based format, to store rich social network data. No query or transformation issues are addressed in these works.

The problem of integrating and standardizing SN applications at a reasonable level of abstraction remains an open question [18]. Mika [26, ch. 5] discusses representation models, tools and standards from the SNA, and Semantic Web communities, and reaches the same conclusion that current tools and data formats do not solve data representation and aggregation requirements. He proposes a solution based on ontologies and automated reasoning. Erétéo et al. [14] present a framework for "semantic aware social network analysis". Also Jung and Euzenat [22] propose a model to represent and extract information from social networks. From a data management point of view, none of these models cope with the requirements. There are many works that extract, model, manipulate, and analyze social

[2] `http://nwb.slis.indiana.edu/`

networks in the Semantic Web for specific applications, for example, that by Aleman-Meza et al. [3] (semantic web application for conflict of interest detection in the context scientific peer review), Kinsella et al. [23], and Mika [25].

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43(1):62–124, 1991.

[3] Boanerges Aleman-Meza, Meenakshi Nagarajan, Cartic Ramakrishnan, Li Ding, Pranam Kolari, Amit P. Sheth, I. Budak Arpinar, Anupam Joshi, and Tim Finin. Semantic analytics on social networks: Experiences in addressing the problem of conflict of interest detection. In *WWW 2006*, pages 407–416, 2006.

[4] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Cong Yu. Socialscope: Enabling information discovery on social content sites. In *CIDR*. www.crdrdb.org, 2009.

[5] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[6] H. Blau, N. Immerman, and D. Jensen. A visual language for querying and updating graphs. Compueter Science Technical Report 2002-037, University of Massachusetts, Amherst, 2002.

[7] Ulrik Brandes and Thomas Erlebach, editors. *Network Analysis: Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004]*, volume 3418 of *Lecture Notes in Computer Science*. Springer, 2005.

[8] Luca Cabibbo. The expressive power of stratified logic programs with value invention. *Inf. Comput.*, 147(1):22–56, 1998.

[9] Kathleen M. Carley. Linking capabilities to needs. In Ronald Breiger, Kathleen M. Carley, and Philippa Pattison, editors, *Dynamic So-*

cial Network Modeling and Analysis: Workshop Summary and Papers, pages 363–370. The National Academies Press, 2003.

[10] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *PODS*, pages 404–416. ACM Press, 1990.

[11] Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in graphlog and datalog. In Serge Abiteboul and Paris C. Kanellakis, editors, *ICDT*, volume 470 of *Lecture Notes in Computer Science*, pages 379–394. Springer, 1990.

[12] Wouter de Nooy, Andrej Mrvar, and Vladimir Batagelj. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2005.

[13] Anton Dries, Siegfried Nijssen, and Luc De Raedt. A query language for analyzing networks. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin, editors, *CIKM*, pages 485–494. ACM, 2009.

[14] Guillaume Erétéo et al. A state of the art on social network analysis and its applications on a semantic web. In *SDoW2008 at ISWC 2008*, 2008.

[15] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Composing schema mappings: Second-order dependencies to the rescue. *ACM Trans. Database Syst.*, 30(4):994–1055, 2005.

[16] Jim Gray, David T. Liu, María A. Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[17] R. Guting. Graphdb: modeling and querying graphs in databases. In *20th VLDB Conference*, pages 297–308, 1994.

[18] Harry Halpin. Beyond walled gardens: Open standards for the social web. In *SDoW2008 at ISWC2008*, 2008.

[19] Mark S. Handcock, David R. Hunter, Carter T. Butts, Steven M. Goodreau, and Martina Morris. statnet: Software tools for the representation, visualization, analysis and simulation of network data. *Journal of Statistical Software*, 24(1):1–11, 2 2008.

[20] H. V. Jagadish and Frank Olken. Database Management for Life Science Research: Summary Report of the Workshop on Data Management for Molecular and Cell Biology at the National Library of Medicine, Bethesda, Maryland, February 2-3, 2003. *OMICS*, 7(1):131–137, 2003.

[21] David Jensen and Jeniffer Neville. Data mining in social networks. In *Dynamic Social Network Modeling and Analysis*, pages 289–302, 2003.

[22] Jason J. Jung and Jérôme Euzenat. Towards semantic social networks. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *ESWC*, volume 4519 of *LNCS*, pages 267–280. Springer, 2007.

[23] S. Kinsella, A. Harth, A. Troussov, M. Sogrin, J. Judge, C. Hayes, and J. G. Breslin. Navigating and annotating semantically-enabled networks of people and associated objects. In *The 4th Conference on Applications of Social Network Analysis (ASNA 2007)*, September 2007.

[24] Stefan Klink, Patrick Reuther, Alexander Weber, Bernd Walter, and Michael Ley. Analysing social networks within bibliographical data. In *DEXA 2006*, pages 234–243, 2006.

[25] Peter Mika. Social networks and the semantic web. In *Web Intelligence*, pages 285–291. IEEE Computer Society, 2004.

[26] Peter Mika. *Social Networks and the Semantic Web*, volume 5 of *Semantic Web And Beyond Computing for Human Experience*. Springer, 2007.

[27] Royi Ronen and Oded Shmueli. Soql: A language for querying and creating data in social networks. In *ICDE*, pages 1595–1602. IEEE, 2009.

[28] Mauro San Martín and Claudio Gutierrez. Representing, querying and transforming social networks with RDF/SPARQL. *ESCW2009*, 2009.

[29] Maksim Tsvetovat, Jana Diesner, and Kathleen M. Carley. Netintel: A database for manipulation of rich social network data. *CMU-ISRI-04-135*, March 2005.

[30] Maksim Tsvetovat, Jeff Reminga, and Kathleen M. Carley. Dynetml: Interchange format for rich social network data. *CMU-ISRI-04-105*, 2004.

[31] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146. ACM, 1982.

[32] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, first edition, 1994.