# SNQL: A Social Networks Query and Transformation Language

Mauro San Martín[1], Claudio Gutierrez[2], and Peter T. Wood[3]

[1]Dept. of Mathematics, U. de La Serena - [2]Dept. of Computer Science, U. de Chile
[3]Dept. of Computer Science and Information Systems, Birkbeck, Univ. of London

**Abstract.** Social Network (SN) data has become ubiquitous, demanding advanced and flexible means to represent, transform and query such data. In addition to the intrinsic challenges of querying graph data is the requirement that networks be restructured, and thus that new values be created.
To address these, we introduce a dedicated data model and query language, SNQL, founded on previous research on graph databases and on the experience of SN researchers. Technically, it is based in GraphLog and second-order tuple generating dependencies, allowing expressiveness for graph querying and node creation as required by SN, while keeping the complexity of query evaluation in NLOGSPACE.

## 1 Introduction

The widespread embracing of social networking services—such as Facebook, MySpace and LinkedIn—as the indispensable tools to manage people's digital and physical lives has resulted in rapidly growing amounts of social activity data. Furthermore, the social information represented by many sites usually includes not only people but a great variety of objects (usually referred to generically as *actors* in the social networking literature) and relations [4]: photographs (Flickr), other sites (del.icio.us), places (Yahoo! Travel), goods (Amazon), and so on. This diversity produces complex social networks (SN) which require managing, querying and transforming.

In the context of these new applications, users have access to an increasing amount of SN data, and consequently the need arises for them to manage and build their own networks based on relevant "pieces" of the huge networks available. For example, it is common to find applicable online information about published research linked with departmental and local scientific networks. This calls for more advanced and flexible means to query and transform SN data. Along the same lines, scientific experimentation with social network analysis (SNA) tools (e.g. Pajek [10]) calls for data management tools to extract parts of SN from different environments and to integrate, filter and transform them.

A further requirement is that SN data representation needs to be flexible enough to incorporate on-the-fly attributes (e.g. for data curation). In addition, data is used and seen from diverse points of view by different users. Hence classical modeling in terms of a fixed set of entities, attributes and relationships does not work well. For example, in SNA it is common for attributes to become

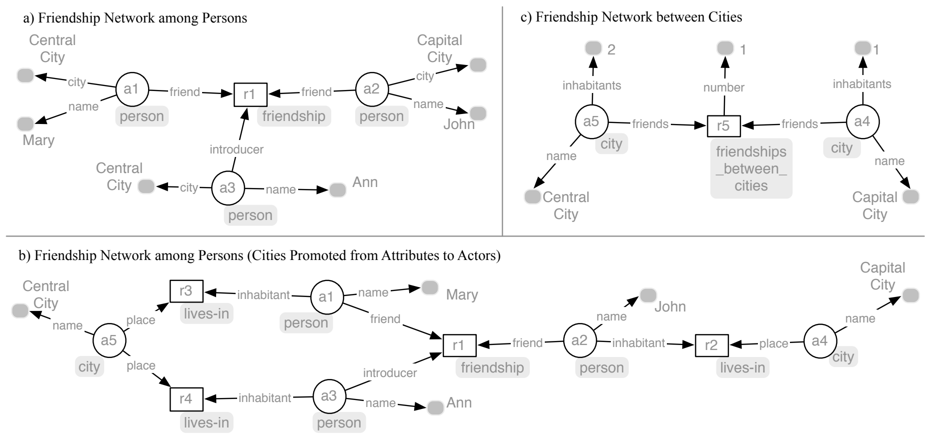

**Fig. 1.** *Friendship Network and Transformations.* a) A social network representing the friendship relation (square node) between *Mary* and *John*, who were introduced by *Ann* (actors as round nodes, and attribute values as grey dots). b) The same social network after promoting *city* attributes 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, for aggregated data of actors to become attributes, and for relations to have arity greater than two.

*Example 1.* Consider a SN of friendship relations among people, including which other person, if any, 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. 1(a)). The study of the relevance of city of residence to friendship might require promoting cities to actors and linking people and cities with a new type of relation, e.g. 'lives-in' (see Fig. 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. 1(c)). □

The requirements suggested by Example 1 together with the integration of diverse types of information as described above call for a simple and flexible model of data for SN. Furthermore, due to the growing volume of SN data, any transformation and query language for SN should be scalable. This implies that a transformation and query language should conform, from a theoretical perspective, to low complexity bounds, and, because of practical concerns, be simple and modular while being sufficiently expressive.

Several of these challenges have been already voiced. Fifteen years ago Freeman defined the *maximal structure experiment* that extended the basic network representation to include attributes as well as to accommodate changes over time [13]. More recently, the need to improve both network data formats in the context of the social web as well as data management services for large and dynamic social networks has been identified [8, 17].

To date, the above challenges have been addressed with ad-hoc approaches, and to the best of our knowledge there is no generic data management solution in spite of the wide agreement on the urgent necessity of addressing this problem [16, 20]. In the appendix on related work we will review widely-used tools in SNA, such as Pajek and UCINET [15], which focus on the final analysis of SN data. Some proposals for graph databases (e.g. GraphDB [14]) have features to deal with SN data, but with ad-hoc developer-oriented languages. In the same spirit some SN services provide APIs (e.g. Facebook's Graph API[1] and the Open Graph Protocol).

The three most comprehensive proposals on the challenges presented above are BiQL, SocialScope, and SoQL. *SocialScope* [3] is a logical architecture for discovering and managing social information, which includes an algebraic query language. It does not provide the capability to construct new data nor deal with the complexity of data dynamics (particularly transformation of actors into attributes). *SoQL* [18] is an SQL-like query language for SN, focused on identifying groups and paths over a classical network. It does not incorporate data transformation features. Finally, *BiQL* [11] is also an SQL-like language with a rich set of features. However, efficiency is still not a concern in these proposals, and none of them analyze computational complexity aspects of the language.

Our *transformation and query language*, named SNQL, is inspired by two earlier languages: GraphLog [9] and second-order tuple-generating dependencies (SO tgds) [12]. SNQL comprises, both syntactically and conceptually, two modules, one for SN matching and one for SN construction, which essentially correspond to GraphLog and SO tgds, respectively.

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.

*Example 2.* Consider again the SN from Example 1. The following SNQL query produces the network depicted in Fig. 1(b) from that in Fig. 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, denote an *extraction pattern* and a *construction pattern*, respectively. `FriendshipNetwork` is the SN shown in Fig. 1(a).

[1] http://developers.facebook.com/docs/reference/api/

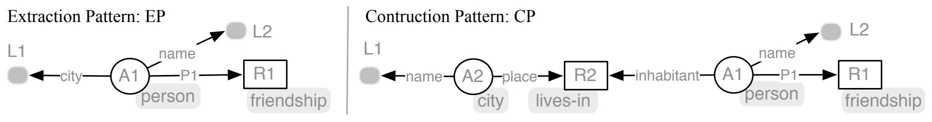

**Fig. 2.** *Attribute Promotion.* Patterns `EP` and `CP` transform attribute 'city' to a new actor whose id is functionally created from its literal value (bound to variable `L1`). Also new relations are required to link persons to the newly created city actors.

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)`. □

Creation of values was addressed by database query languages such as IQL [2] and ILOG [7], that allowed oid or value invention. By relying on rules that use both recursion and oid invention, it can be shown that such languages can express all computable database queries [7]. However, it turns out that when recursion and invention are not allowed to interact, as in our proposal, queries can be evaluated in PTIME [2]. In our language, we consider another formalism used to invent values, namely, SO tgds [12]. SO tgds use existentially quantified function symbols (and equalities) to specify the composition of schema mappings, where values in a target schema (output) may need to be existentially quantified (invented). Since SO tgds are not recursive, materializing the result of a schema mapping (which corresponds to creating an SN in our setting) can be done in PTIME [12].

Our contributions are as follows:

1. A data structure capable of representing the informational richness and malleability of social networks.
2. A transformation and query language satisfying the data management requirements of social networks with good properties: adequate expressiveness, and accessible to social networks field practitioners.
3. A query language that includes object creation but maintains low complexity.
4. A corresponding evaluation algorithm whose complexity scales adequately.

The outline of the rest of the paper is as follows. Section 2 briefly presents the requirements and SNA use cases. Section 3 defines the syntax (via examples) and semantics of SNQL. Section 4 presents results on the expressiveness and complexity of SNQL. We included an Appendix with complete syntax and extended related work.

## 2 SNQL Requirements and Use Cases

To identify the general requirements and operations needed for SNQL, we collected use cases and archetypical operations from standard SNA literature [22,

**Table 1.** Selected social network data management use cases.

| Use Case | Description | Required Query Features |
|---|---|---|
| 1. Selecting Groups | Select a subnetwork of actors and relations that satisfy conditions on their attribute values and/or participation in certain relations. | Pattern matching, filtering by attributes values. |
| 2. Promoting Attributes to Actors | From an actor $A_1$ and one of its attributes $(att, v)$ produce a new actor $A_2 = f(v)$ and a new relation $R = g(A_1, v)$ ($f$ and $g$ functions): all actors with value $v$ for $att$ will be connected to $A_2$. | Pattern matching, creation of new objects, pattern production. |
| 3. Identifying Brokers | Each time a characteristic brokerage pattern is found, label the broker in the output accordingly. Some brokerage patterns require that certain relations do not exist. | Pattern matching, negation, pattern production. |
| 4. Counting Binary Relations | Select all relations of a given type, group by participant actors, count. Produce only one relation per group with the new attribute count. | Pattern matching, aggregation, pattern production. |
| 5. Ego-network selection | Select an actor along with all its direct neighbors, and the relations between them. | Pattern matching, induced subgraphs. |
| 6. $k$-neighborhood | Same as above but to distance $k$ instead of one. | Pattern matching, transitive closure, induced subgraphs. |

19, 10], surveyed relevant publications, mainly from the journal *Social Networks*, and studied the operations available in SNA software tools such as Pajek [10].

### 2.1 Data structure requirements and definition

The natural and traditional choice for representing social networks is to use graphs where actors are nodes and relations are edges. However, doing so limits the representation power to that of binary relations and forbids attributes on relations. Thus, our logical data structure, the social networks data model (SNDM), is a graph where actors, relations and attributes are all modeled as nodes, and edges associate attributes with the actors or relations they describe, and actors with the relations in which they participate. This structure is implemented using three sets of triples:

– A typing set N. Each triple $(oid, [isa|isr], family) \in N$ indicates that the actor or relation *oid* belongs to (is of type) *family*.
– A set $R$ indicating roles. Each triple $(a\_oid, role, r\_oid) \in R$ indicates that the actor *a_oid* participates with *role* in the relation *r_oid*.
– A set $M$ describing attributes. Each triple$(oid, pred, v) \in M$ indicates that the actor or relation *oid* has the attribute *pred* with value $v$.

### 2.2 Requirements of SNQL

Table 1 summarizes the use cases from the list gathered from the sources mentioned above. Each use case is selected for its relevance and justifies the inclusion of a query language feature.

Social networks operations can be divided into two groups: *data management* operations that return a social network, and *measure* operations that return values or sets of values, such as centrality. Today there are various tools that deal with measure operations (e.g. Pajek, R, and UCINET) and clearly belong more to the SNA tools field than to the data management one.

SNQL focuses on the first type, data management operations, which produce networks from networks. Through the use of aggregate functions, we assume that node, group and network measures are available to the language but do not need to be expressible in the language itself. Brandes [5] offers a survey of such measures and the corresponding evaluation algorithms.

Each SNQL query must be able to filter and/or transform a given SN into a new SN. Filter queries are used to reduce the size of a SN and to focus on relevant groups. Transformations produce a new SN where some implicit structural element has been made explicit.

## 3 The SNQL Query Language

The design of SNQL addresses the following issues: to be expressive enough, and keep the evaluation cost under practical bounds. The challenge was to identify a few generic operations to cover the required expressiveness, and to be closed, that is, to be able to construct (and transform) social networks into social networks. The solution was to use GraphLog and second-order tuple generating dependencies, allowing expressiveness for graph querying (see Figure 3).

### 3.1 Query Syntax

The language should be friendly enough for both the lay-user and the programmer. For the former, 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 [9]. For the latter, 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, 9], and the data exchange formalism called second-order tuple generating dependencies [12].

An SNQL construct query $Q$ follows the standard SELECT|CONSTRUCT – WHERE – FROM structure of languages like SQL and SPARQL. It receives as

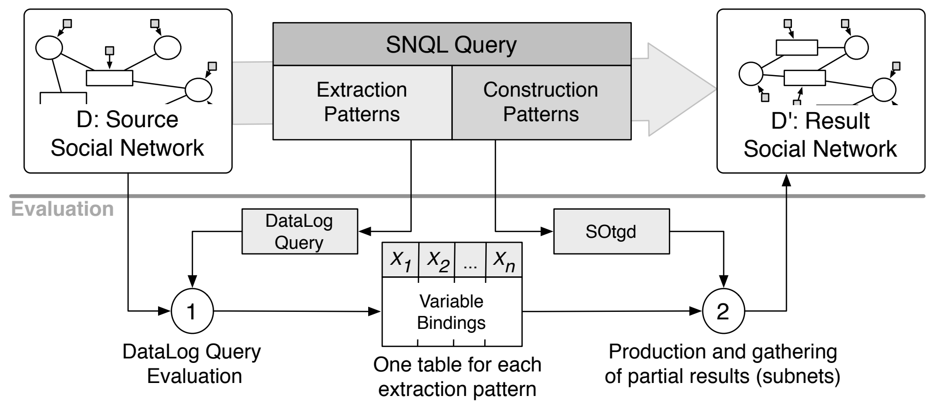

**Fig. 3.** *Overview of the SNQL language.* An SNQL query is composed of an extraction (datalog-like) plus a construction (SO tgd-like) pattern. The evaluation process has two stages: (1) the Datalog program is evaluated over the data source network $D$ producing intermediate tables of variable bindings; (2) partial results are produced from the obtained tuples (a table for each distinguished predicate) using the SO tgd. All partial results are gathered in the final result network $D'$.

input social networks (the FROM clause), extracts information using patterns (the WHERE clause), and outputs a new social network, possibly with new values, using the CONSTRUCT clause. For space reasons, we present the syntax of SNQL by example, using the SN of friendship relations among people presented in Example 1 (see Fig. 1) (the complete syntax is in Appendix A).

*Example 3.* (Promoting Attributes to Actors) Recall the query from Example 2, where the patterns EP and CP were depicted in Fig. 2. Expanding the patterns as lists of triples, the query is as follows:

```
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
```

*Example 4.* (Grouping and Aggregation) The following SNQL query produces the network depicted in Fig. 1(c) from that in Fig. 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)
  FROM    FriendshipNetwork
UNION
CONSTRUCT CP2 IF A5 = f(L2) AND A6 = f(L3) AND R2 = g(A5, A6) AS SN2
```

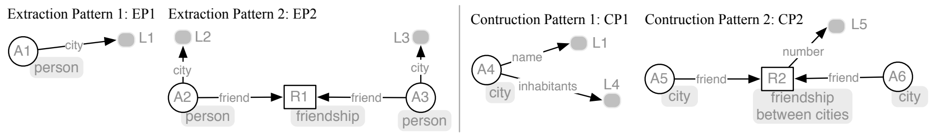
Extraction Pattern 1: EP1
Extraction Pattern 2: EP2
Contruction Pattern 1: CP1
Contruction Pattern 2: CP2
A1
city
L1
person
L2
L3
A2
friend
R1
friendship
A3
person
A4
name
inhabitants
L4
A5
R2
number
L5
A6
friendship between cities

**Fig. 4.** *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.

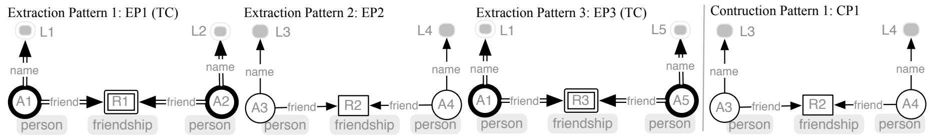
Extraction Pattern 1: EP1 (TC)
Extraction Pattern 2: EP2
Extraction Pattern 3: EP3 (TC)
Contruction Pattern 1: CP1
L1
L2
L3
L4
L5
name
A1
friend
R1
A2
person
friendship
A3
R2
A4
R3
A5

**Fig. 5.** *Transitive Closure.* Patterns `EP1` and `EP3` identify all transitively reachable actors from person named 'John' through 'friendship' relations. Patterns `EP2` and `CP1` produce the relations induced by pairs of actors in the set of reachable actors.

```
  WHERE   AGG({L2,L3}, COUNT AS L5, EP2 FILTER (L2 != L3))
  FROM    FriendshipNetwork
```

Patterns `EP1`, `EP2`, `CP1`, and `CP2` are depicted in Fig. 4. 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. □

*Example 5.* (Transitive Closure) The following SNQL query (whose patterns are shown in Fig. 5) produces the network comprising an actor that meets a given criterion (his name is 'John'), along with all other actors that can be reached transitively by matching the given pattern (of friendship relations). Additionally all induced relations between pairs of reachable actors are included in the result.

```
CONSTRUCT CP1
  WHERE   EP2 FILTER ((A3 != A4) AND (A3 = A1 OR A3 = A2) AND
                      (A4 = A1 OR A4 = A5))
          AND (TC(A1, A2, EP1) WITH L1='John')
          AND (TC(A1, A5, EP3) WITH L1='John')
  FROM    FriendshipNetwork
```

`TC` returns the transitive closure of the binary relation formed by all instantiations of the variables appearing as its first and second arguments when matching the extraction pattern of its third argument. A starting condition is specified after `WITH`. As a result, variables `A2` and `A5` are bound to the people reachable

0. Each triple `t` of the form `(A,B,C)` is translated as $t(A, B, C)$, where $t$ is $n, r$ or $m$ according to the type of the triple.
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`: $p(\overline{z}) \leftarrow p_1(\overline{x}), \neg p_2(\overline{y})$.
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 $\mathbb{A}$)

**Fig. 6.** Translation of Extraction Pattern to Datalog.

from John through transitive friendship relations. Pattern `EP2`, along with the `FILTER` conditions above, is then used to match all the induced friendship relations between distinct pairs of people reachable from 'John' (along with 'John' himself). □

### 3.2 Query Semantics

An SNQL query $Q$ of the form `CONSTRUCT <T> WHERE <PATT> FROM <S>` transforms social networks into social networks, and is evaluated as shown in Figure 3.

The formal semantics can be expressed in standard formalisms (Datalog and tuple-generating dependencies) as follows. Let $D$ be a social network, $Q$ an SNQL query, and $Q(D)$ the result of applying $Q$ to $D$. For set of variables $X$, let $\overline{x}$ be the tuple comprising all variables in $X$.

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` 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 Fig. 6.

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.

```
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
```

**Fig. 7.** Translation of query in Example 4 to Datalog.

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:

$$construct(v_1, \ldots, v_k) \leftarrow p(\overline{z}) \wedge \bigwedge_j eq_j. \tag{1}$$

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

$$SN = \bigcup \left\{ t(u_1, u_2, u_3) : \exists(..u_1..u_2..u_3..) \in \mathit{construct} \text{ and } t \text{ in } \texttt{trList} \right\} \tag{2}$$

*Example 6.* Consider the SNQL query $Q$ of Example 4 that involved grouping and aggregation. The translation of $Q$ to Datalog is shown in Figure 7. □

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 (see Section 4).

## 4 Complexity and Expressiveness

The main goal of this paper was to introduce a sufficiently flexible data model and expressive query language that meets the data manipulation requirements of social networks. In this section we state—without proof due to space constraints—results that show the good behaviour of the language regarding complexity and expressive power.

SNQL is composed of two modules: one for extraction of information and one for construction of 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, 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 [9]. 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 [12].

Knowing this, it is not surprising that SNQL covers all use cases 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 cohesive subgroups defined using 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 LOGSPACE 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 1 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. For the induced subgraph, Example 5 gives the idea how this is done.

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.*

**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:*

$$(p(\overline{x}) \wedge \bigwedge_k eq_k) \rightarrow (t_1 \wedge \ldots \wedge t_r), \tag{3}$$

(a) For an expression
`CONSTRUCT trList1 IF eqList1; ... trListn IF eqListn;`
translated in the form of eq. (3), define $n$ clauses: $aux_j(\overline{x_j}) \leftarrow p(\overline{x}) \wedge \bigwedge_k eq_k, j = 1, \dots, n$.

(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 Datalog program generated from the extraction pattern (see Fig. 6).

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

**Fig. 8.** Evaluation Algorithm.

*where $p(\overline{x})$ and $eq_k$ follow the notation of equations (1) and (2), that is, predicate $p(\overline{x})$ is the result of the processing of the extraction pattern, and $t_j$ and $eq_k$ are predicates resulting from the translation of the triples and equalities in the CONSTRUCT clause, and each tuple $\overline{x_i}$ includes all variables in $p$ and in the $eq_k$'s.*

A naive implementation of the semantics presented in Fig. 6 would materialize intermediate results.

This can be avoided by using the algorithm in Fig. 8.

**Lemma 1.** *The Evaluation Algorithm is correct.*

It is possible to show that the above evaluation computes queries efficiently from a database perspective. As is customary when studying the complexity of the evaluation problem for a query language [21], 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.*

## 5 Conclusions

Based on social network practice, we have presented the design of a data model and query language for SN. Of particular novelty, is the ability of our language to transform one network into another, in the process creating new actors and new attributes based on aggregation, features crucial for social network researchers.

We presented the syntax, semantics and complexity analysis. The language is based on classical database results to obtain a good balance between expressiveness and complexity. Presenting both a graphical and SQL-like syntax, we designed it to have the most encompassing expressiveness while staying tractable. In fact, we show that it includes all tractable operations found in our survey of SN data management practice, and we prove that the cost of transforming networks can be done efficiently from a computational point of view. all tractable operations found in our survey of current SN data management practice

## References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Abiteboul, S., Kanellakis, P.C.: Object identity as a query language primitive. J. ACM 45(5), 798–842 (1998)
3. Amer-Yahia, S., Lakshmanan, L.V.S., Yu, C.: Socialscope: Enabling information discovery on social content sites. In: CIDR. www.crdrdb.org (2009)
4. Amer-Yahia, S., Markl, V., Halevy, A.Y., Doan, A., Alonso, G., Kossmann, D., Weikum, G.: Databases and web 2.0 panel at vldb 2007. SIGMOD Record 37(1), 49–52 (2008)
5. Brandes, U., Erlebach, T. (eds.): Network Analysis: Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004], Lecture Notes in Computer Science, vol. 3418. Springer (2005)
6. Breiger, R., Carley, K., Pattison, P. (eds.): Dynamic Social Network Modeling and Analysis: Workshop Summary and Papers. The National Academies Press (2003)
7. Cabibbo, L.: The expressive power of stratified logic programs with value invention. Inf. Comput. 147(1), 22–56 (1998)
8. Carley, K.M.: Linking capabilities to needs. In: Breiger et al. [6], pp. 363–370
9. Consens, M.P., Mendelzon, A.O.: Graphlog: a visual formalism for real life recursion. In: PODS. pp. 404–416. ACM Press (1990)
10. de Nooy, W., Mrvar, A., Batagelj, V.: Exploratory Social Network Analysis with Pajek. Cambridge University Press (2005)
11. Dries, A., Nijssen, S., Raedt, L.D.: A query language for analyzing networks. In: Cheung, D.W.L., Song, I.Y., Chu, W.W., Hu, X., Lin, J.J. (eds.) CIKM. pp. 485–494. ACM (2009)
12. Fagin, R., Kolaitis, P.G., Popa, L., Tan, W.C.: Composing schema mappings: Second-order dependencies to the rescue. ACM Trans. Database Syst. 30(4), 994–1055 (2005)
13. Freeman, L., Romney, A.K., White, D.R. (eds.): Research Methods in Social Network Analysis. Transaction Publishers (1992)
14. Güting, R.: Graphdb: modeling and querying graphs in databases. In: 20th VLDB Conference. pp. 297–308 (1994)
15. Huisman, M., van Duijn, M.: Software for Social Network Analysis, chap. Software for Social Network Analysis, pp. 270–316. Cambridge (2005)
16. Jagadish, H.V., Olken, F.: Database management for life science research: Summary report of the workshop on data management for molecular and cell biology. OMICS 7(1), 131–137 (2003)
17. Mika, P.: Social Networks and the Semantic Web, Semantic Web And Beyond Computing for Human Experience, vol. 5. Springer (2007)
18. Ronen, R., Shmueli, O.: Soql: A language for querying and creating data in social networks. In: ICDE. pp. 1595–1602. IEEE (2009)
19. Scott, J.: Social Network Analysis. SAGE Publications, second edn. (2000)

20. Topaloglou, T., Davidson, S.B., Jagadish, H.V., Markowitz, V.M., Steeg, E.W., Tyers, M.: Biological data management: Research, practice and opportunities. In: VLDB. pp. 1233–1236 (2004)
21. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: STOC. pp. 137–146. ACM (1982)
22. Wasserman, S., Faust, K.: Social Network Analysis: Methods and Applications. Structural Analysis in the Social Sciences, Cambridge University Press (1994)

## A SNQL Syntax

```
<construct-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>
```