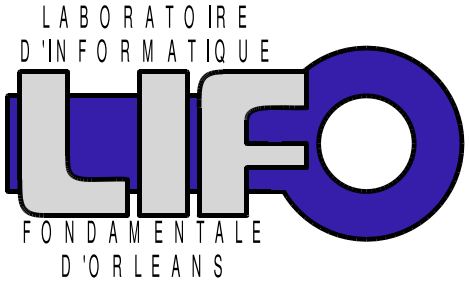
Urban Graph Analysis on a Context-driven Querying System

Jacques Chabin, Luiz Gomes-Jr, Mirian Halfeld-Ferrari



4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Urban Graph Analysis on a Context-driven Querying System

Jacques Chabin, Luiz Gomes-Jr.,
Mirian Halfeld Ferrari
LIFO, Université d'Orléans

Rapport n^o **RR-2018-01**

Urban Graph Analysis on a Context-driven Querying System

JACQUES CHABIN, Université d'Orléans, INSA CVL, LIFO EA
LUIZ GOMES-JR., DAINF, UTFPR
MIRIAN HALFELD-FERRARI, Université d'Orléans, INSA CVL, LIFO EA

Urban computing intends to provide guidance to solve problems in big cities such as pollution, energy consumption, and human mobility. These are, however, difficult problems with several variables interacting in complex patterns. Such complex systems are often represented as graphs, taking advantage of the flexibility of the model and the availability of network science tools for analysis. In this context, expressive query languages capable of tackling graph analysis problems on a customized context are essential.

This paper presents a context-driven query system for urban computing where users are responsible for defining their own restrictions over which datalog-like queries are built. Instead of imposing constraints on databases, our goal is to filter consistent data during the query process. Our query language is able to express aggregates in recursive rules, allowing to it capture network properties typical of graph analysis. This paper presents our query system and analyzes its capabilities using use cases in Urban Computing.

1 INTRODUCTION

Urban computing deals with integrated data acquired from a variety of traditional data sources as well as from sensors and other devices [12]. A graph-based data model is usually adopted for representing datasets involved in urban computing, from traffic and points of interest, to more conventional data, such as population statistics. Therefore, planning and solving problems in cities' context rely on analysing and extracting knowledge from graphs which require the ability to compute graph queries such as reachability or shortest paths, or, more generally, queries imposing a repeated computation over the graph until reaching a threshold or fixpoint.

Defining a declarative, powerful query language capable of expressing useful queries in this context is still a challenge. Datalog (a recursive and declarative query language) has been proposed as an option for data scientists to succinctly describe iterative graph analytics (see, for instance [2, 22]). Possible datalog extensions allow group-by and aggregate queries which are the basis for building more sophisticate computations such as Page Rank [8].

Manuscript submitted to ACM

Disastrous results can be obtained from decisions made on the basis of insights built over invalid data. In other words, if the data being analysed is not accurate, the resulting analysis is misleading. Indeed as the "GIGO (Garbage In, Garbage Out) principle is not sustainable" [24], ensuring graph analysis accuracy becomes a major issue; stressing the importance of data quality, initially neglected in the big data context in the name of velocity and volume.

Indeed, answering a user's query means offering him appropriate and accurate results, taking into account his needs and his context. Constraints have always been a tool for specifying the requirements of data consistency (based on semantic, shape or context restrictions). They can naturally be used as the instrument of query personalization and user's quality exigences.

This paper merges the validity-checking-while-querying process proposed in [9, 10] with operators introduced in [14] and built to allow graph analysis. In this way, the paper presents a declarative query language capable of expressing graph analytic queries, *i.e.,* iterative or recursive queries whose semantics involves constraint checking together with recurrent computation. Thus, it generalizes the previous approaches in [9, 10, 14] focusing on the semantics of this query language extension which ensures validity *w.r.t.* a given set of constraints.

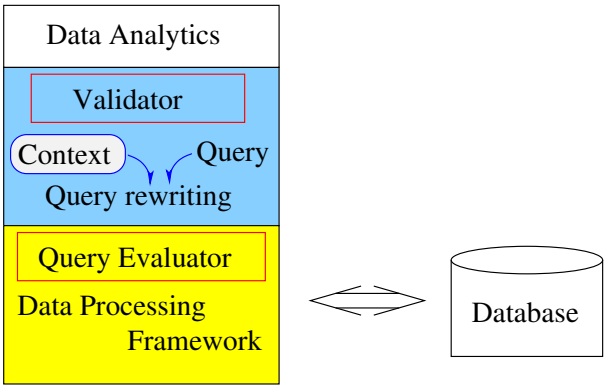


Fig. 1. General System Architecture.

Figure1 illustrates a general system architecture where our validator appears as a top (blue) layer over a query-evaluator. The validator filters valid answers *w.r.t.* a given context in two ways discussed in [9]: (1) by rewriting queries *w.r.t.* this context or (2) when the evaluator has insufficient capabilities to evaluate the rewritten query, by establishing a dialogue between validator and evaluator via subsequent sub-queries. Either way, the validator ensures that only valid answers are sent to the user. This architecture can be associated to the cloud data management software stack sketched in [7], where distributed storage is envisaged. Indeed, as a top layer, our validator can work with different query evaluators, connected to different mediators to which it adds its filter capabilities.

The current paper focuses on proposing a query language allowing graph analytic queries whose semantics rely on validity *w.r.t.* personalised contexts. Thus, to highlight our goal, we ignore, here, all the considerations concerning query evaluation on a (distributed or centralized) database. We assume that query answers are generated by a query evaluator which, for this paper, is simulated by an in-memory program.

Our work enables a query environment where users specify the context on which queries should be processed. A user sets his *customised context* by specifying the (integrity) constraints he imposes on data. In this way, data quality requisites are placed closer to the data consumers, and become "everyone's business"[24].

Our main contribution is the integration of two important concepts dealt separately in other proposals: (i) constraint checking (which enables context-aware queries) and (ii) query expressiveness for graph analysis (which enables recursive computations over networks). In this paper, we show the interplay between these two concepts through practical use cases.

To reach our goal, the aforementioned proposals have been extended, here, in the following aspects: (i) the non-recursive datalog-like query language proposed in [9, 10] is enhanced to allow *recursive datalog programs* as queries which may, now, contain group-by or aggregate sub-queries; (ii) operators proposed in [14] are expressed through a datalog program, offering a declarative and well-defined mechanism to explain their semantics and (iii) the query process proposed in [14] includes now a mechanism for *filtering* answers according to the user's context. Even if data sources are not consistent *w.r.t.* the given constraints, our system can render the consistent answers (*i.e.,* those which are computed with data that do not violate any constraint).

Paper Organization: Section 2 illustrates how a querying environment which integrates constraint checking with a declarative query language can be useful on an urban computing environment. Section 3 offers preliminaries for the formalization of our approach while Section 4 introduces the constraints considered in our work. The semantics of our recursive datalog-like query language allowing constraint checking is presented in Section 5. Section 6 presents an use case. Section 7 discusses some related work while Section 8 concludes our paper.

2 URBAN COMPUTING: QUERIES AND CONTEXTS

As an interdisciplinary field, urban computing deals with many different questions, trying to improve the quality of densely populated areas, including decision aid to government leaders. In this context, queries involving transport and strategic placement decisions are usual.

As an example, let us consider the means of transportation in a city like Paris. Figure 2 presents an extract of the transportation network while Table 1 shows some queries that we will be discussing in this paper. The transportation network is a graph where each node has the name of a specific place where a user can find a means of transport (a Metro station, a bus stop, a *Batobus* stop, a *Velib* station, etc). Relation Transp stores these networks. Besides attributes indicating an edge (X_{to}, X_{from}) of the transportation graph, the relation stores information about the means of transport (X_{means} – bus, underground, boat, bike, etc), the estimated travel time (X_{time}) and carbon footprint (X_{cfp}) for the graph's edge. The network is thus a multi-graph; also seen as a superposition of different graphs (one for each means of transport). Relation Environment stores the facilities (gardens, hospitals, clinics, car parks, station, restaurants, hotels, . . .) around a place $X_{placeName}$. Other relations contribute with details of data stored in Transp and Environment.

Queries on such a graph can vary from a simple '*Is it possible to go from Alesia to Montparnasse by tram?*' to graph analysis such as the minimal path to go from *Gare d'Austerlitz* to *Saint-Lazare* or the best place for a new public crèche, according to governmental constraints.

The challenge in proposing a new querying process for graphs is not only on the expression power of a declarative system, but also in ensuring data quality. Systems adapting query languages to interact with constraint checkers represent an important step towards monitoring data quality at scale. Our work fits in this perspective and proposes a querying environment where answers are filtered according to a given context. The context personalises the query environment, establishing a set of constraints to be respected. Validity of data is therefore defined *w.r.t.* a context. Inconsistency on source databases is possible but answers are filtered and only valid information is returned to the user.

$\mathsf{Connexion}(X_{from}, X_{to}, X_{time}, X_{cfp})$	$\leftarrow$	$\mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{cfp})$.
$\mathsf{Connexion}(X_{from}, X_{to}, X_{time}, X_{cfp})$	$\leftarrow$	$\mathsf{Transp}(X_{from}, Z, Z_{means}, Z_{time}, Z_{cfp})$, $\mathsf{Connexion}(Z, X_{to}, Y_{time}, Y_{cfp})$, $X_{time} = Z_{time} + Y_{time}, X_{cfp} = Z_{cfp} + Y_{cfp}$.
$\mathsf{Q}_0(Alesia, X_{to}, X_{cfp})$	$\leftarrow$	$\mathsf{Connexion}(Alesia, X_{to}, X_{time}, X_{cfp}), (X_{time} <= 20)$
$\mathsf{Q}_1(X_{from}, X_{to}, a_{totalAvTime}, a_{Avcfp})$	$\leftarrow$	$agg(\mathsf{Connexion}(X_{from}, X_{to}, X_{time}, X_{cfp}), X_{from}, X_{to}, a_{totalAvTime} = AVG(X_{time}),\ \ a_{Avcfp} = AVG(X_{cfp}))$.
$\mathsf{Q}_\beta(X_{from}, X_{to}, c_{cfp})$	$\leftarrow$	$agg(q_{SetRes}(X_{from}, X_{to}, X_{cfp}, i), X_{from}, X_{to}, c_{cfp} = MIN(X_{cfp}))$.

Table 1. Example of queries on a transportation database.

The set of constraints $\mathbb{C}_1$ of context Ctx1

c_1	$\mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{cfp})$	$\rightarrow$	$\mathsf{Type}(X_{means}, Rail)$.
c_2	$\mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{cfp}), (X_{cfp} > 500)$	$\rightarrow$	$\bot$.

Table 2. User's context Ctx1 = $(\mathbb{C}_1, \mathbb{G}_1)$ on a transportation database.

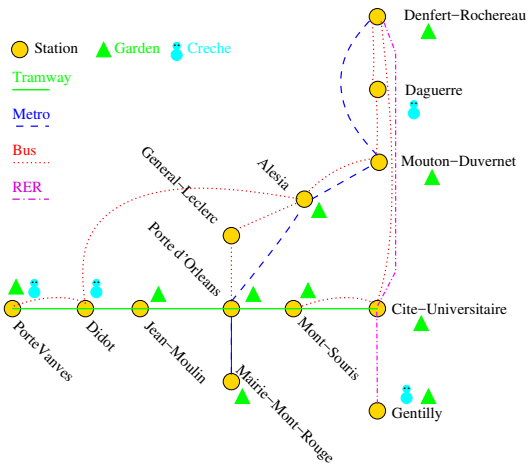


Fig. 2. Example of an urban graph.

In Table 2, constraints $\mathbb{C}_1 = \{c_1, c_2\}$ establish the context Ctx1. Our user is only interested in means of transport using railways (c_1). Moreover, he wants paths composed only by edges associated to a carbon footprint of less then $500cg$ of CO_2 (Table 2, denial constraint c_2, values based on [1]).

In this context, consider the conjunctive query Q_0 (Table 1): starting from *Alesia*, where can we go in less than 20 minutes and what is the carbon print of this path? Q_0 involves three rules. The first two rules compute Connexion, the transitive closure of the graph (*i.e.,* all the paths of a graph). Rule Q_0

just states the maximal time and the starting point of the paths. However, when evaluated on context Ctx1, Connexion is restricted to be the transitive closure of *only* the *rail* transport graph whose edges respect the indicated *carbon footprint limit*. Query re-writing techniques can be used to incorporate user-defined constraints into the body of the query. For instance, our first rule can be re-written as $\mathsf{Connexion}(X_{from}, X_{to}, X_{time}, X_{cfp}) \leftarrow \mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{cfp}), \mathsf{Type}(X_{means}, Rail)$, $(X_{cfp} <= 500)$. Thus, if the first two rules are re-written to incorporate constraints in $\mathbb{C}_1$, then only answers respecting c_1 and c_2 are generated for Q_0. Rewriting algorithms for our conjunctive queries are presented in [9].

All queries evaluated on Ctx1 are restricted in the same way. Let us consider the group-by (or aggregation) query Q_1 which computes the average travel time and average carbon foot print of every two nodes. The body of Q_1 is composed by a conjunctive sub-query involving Connexion whose answers are filtered: only valid tuples *w.r.t.* Ctx1 are taken as input for the aggregate/group-by operators.

As we will see in Section 5.3, the situation is similar for Q_β whose evaluation is based on sub-queries including $q_{follows}(X_{from}, X_{to}, X_{cfp}) \leftarrow \mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{cfp})$ and $q_{set}(Alesia, Alesia, 0, 0)$ whose answers are filtered according to Ctx1 and only then sent as input to other sub-queries taking part in the computation of Q_β's results. In this way, considering $X_{from} = Alesia$, Q_β computes the cfp-minimal rail path between Alesia and any other node in the railway graph.

3 BACKGROUND

This section summarizes some basic notions used in this paper. We refer to [3] for more details on datalog as a query language.

Alphabet. Let **A** be an alphabet consisting of the following pairwise disjoint sets: CONST, a countably infinite set of constant; VAR an infinite set of variables ranging over CONST (we use **X** as an abbreviation to denote the set $\{X_1, \ldots, X_k\}$ where $k > 0$); PRED, a *finite* set of predicates, each associated with its arity.

Atoms. A term is a constant or a variable. An *atomic formula* (or *atom*) has one of the forms: (i) $P(t_1, ..., t_n)$, where P is an n-ary predicate and $t_1, ..., t_n$ are terms; (ii) $\top$ (meaning true) or $\bot$ (meaning false); (iii) $(t_1\ op\ t_2)$, where t_1 and t_2 are terms and op is a comparison operator $(=, <, >, \leq, \geq)$. A *literal* is an atom of the form $P(t_1, ..., t_n)$. An *instantiated* or *ground literal* (also denoted as a *fact*) is an atom of form $P(u)$ where $u \in (\text{CONST})^n$.

Substitution. A *substitution* from the set of symbols E_1 to the set of symbols E_2 is a function $h : E_1 \rightarrow E_2$. A *homomorphism* from the set of atoms A_1 to the set of atoms A_2, both over the same predicate P, is a substitution h from the terms of A_1 to the terms of A_2 such that: (i) if $t \in$ CONST, then $h(t) = t$, and (ii) if $P(t_1, ..., t_n) \in A_1$, then $P(h(t_1), ..., h(t_n)) \in A_2$. If h is a homomorphism, $P(h(t_1), ..., h(t_n))$ is simply denoted by $h(P(t_1, ..., t_n))$. The notion of homomorphism naturally extends to conjunctions of atoms.

Database Schema and Database Instance. A database schema $\mathbb{G}$ is a set of predicates in PRED. A database instance D is a set of *facts* over $\mathbb{G}$.

Datalog. A *(datalog) rule* r is an expression of the form

$$R_0(u_0) \leftarrow R_1(u_1) \ldots R_n(u_n)$$

where $n \geq 0$, R_i $(0 \leq i \leq n)$ are predicate names and u_i are free tuple (*i.e.,* may use either variable or constants) of appropriate arity. The head of r (denoted by $head(r)$) is the expression $R_0(u_0)$ and $R_1(u_1) \ldots R_n(u_n)$ forms the body (denoted by $body(r)$). Rules are *safe range*, *i.e.,* only variables appearing in the rule body can appear in the head. A *datalog program* $\mathbb{P}$ is a finite set of datalog rules. Each predicate occurring only in the body of a rule is an *extensional predicate*. Each predicate occurring in the head of some rule in $\mathbb{P}$ is an *intentional predicate*. The extensional database schema, denoted $edb(\mathbb{P})$, consists of the set of all extensional predicates, whereas the intentional schema $idb(\mathbb{P})$ consists of all intentional ones. The schema of $\mathbb{P}$ is the union of $edb(\mathbb{P})$ and $idb(\mathbb{P})$.

Queries. A (conjuctive) *query* q of arity n over a given schema is a rule whose head defines the output of a datalog program. A *boolean conjunctive* query is a conjunctive query of arity zero.

Query Evaluation. To define the answers of a query, we adopt the semantics of a positive datalog program in the fixpoint perspective, using a bottom-up evaluation strategy. Thus, for a given program $\mathbb{P}$, the evaluation starts with the facts in $edb(\mathbb{P})$ and continues until a fixpoint is reached, *i.e.,* no new facts may be derived for predicates in $idb(\mathbb{P})$. The generation of each new fact is obtained by using the immediate consequence operator whose definition is recalled below.

Definition 3.1 (Immediate consequence operator). Let $\mathbb{P}$ be a datalog program and I be a set of facts. Let T be an operator over $\mathbb{P}$. We denote by ν a homomorphism $\nu : \textsc{var} \Rightarrow \textsc{const}$. We define $T_{\mathbb{P}}(I) = I \cup \{\nu(head(r)) \mid r \in \mathbb{P} \text{ and } \nu(body(r)) \subseteq I\}$. □

The semantics of a datalog program $\mathbb{P}$ on an instance I over $edb(\mathbb{P})$ is defined as the fixpoint of $T_{\mathbb{P}}$, denoted by $T^*_{\mathbb{P}}(I)$. The *answer* to a query q of arity n over I, denoted as $q(I)$, is the set of all n-tuples $t \in \textsc{const}^n$ for which there exists a homomorphism $h_t : var(body(q)) \Rightarrow \textsc{const}$ such that: (1) each literal $L \in h_t(body(q))$ is in $T^*_{\mathbb{P}\cup\{q\}}(I)$, (2) $q(t) = h_t(head(q))$ and (3) $h_t(head(q)) \in T^*_{\mathbb{P}\cup\{q\}}(I)$. Technically, the answer *false* (*i.e.,* a negative answer) for a boolean conjunctive query corresponds to the empty result set and the answer *true* (*i.e.,* a positive answer) corresponds to the result set containing the empty tuple. A positive answer over I is denoted by $I \models q$. A union of conjunctive queries (UCQ) Q of arity n is a set of CQ, where each $q \in Q$ has the same arity n and uses the same predicate symbol in the head. The answer to Q over an instance I, denoted as $Q(I)$, is defined as the set of tuples $\{t \mid \text{ there exists } q \in Q \text{ such that } t \in q(I)\}$.

A datalog program $\mathbb{P}$ is typically viewed as defining a mapping from instances over the *edb* to instances over the *idb*. Our *edb* concerns a (distributed) database, but, as stated in Section 1, details concerning the interaction between the validator and the evaluator are out of the scope of this paper. Therefore, we assume that the global schema on which queries and constraints are built coincides with the database schema being consulted. Moreover, our database instance D summarizes all available information coming from data sources (no matter how it is obtained).

4 CONTEXTS AND CONSTRAINTS

Let $\mathbb{G}$ be a global schema, and let $\mathbb{C}$ be a set of constraints on $\mathbb{G}$. A context Ctx is defined by the pair $(\mathbb{C}, \mathbb{G})$. Queries are defined on $\mathbb{G}$ and evaluated on a database instance D. Valid answers are those respecting context Ctx. Constraints and constraint satisfaction are defined below. Notice that we allow two different constraint formats.

(1) **Positive constraints** ($\mathbb{C}_P$): Each positive constraint has the form

$$\forall \mathbf{X}, \mathbf{Y} \;\; L_1(\mathbf{X}, \mathbf{Y}) \rightarrow \exists \mathbf{Z} \;\; L_2(\mathbf{X}, \mathbf{Z})$$

where L_1 and L_2 are predicate symbols.
(2) **Negative constraints** ($\mathbb{C}_N$): A *negative constraint* is either of the form

$$(\forall \mathbf{X})((comp(\mathbf{X}') \wedge L(\mathbf{X})) \Rightarrow \bot) \quad \text{or} \quad (\forall \mathbf{X})((comp(\mathbf{X}') \wedge L_1(\mathbf{X}_1) \wedge L_2(\mathbf{X}_2)) \Rightarrow \bot) \quad ,$$

where $\mathbf{X}_1 \cap \mathbf{X}_2 \neq \emptyset$ and $comp(\mathbf{X}')$ is a (possibly empty) comparison formula with variables $\mathbf{X}'$ that all occur in $\mathbf{X}$, and where $L(\mathbf{X})$, $L_1(\mathbf{X}_1)$ and $L_2(\mathbf{X}_2)$ are atoms.

The left and right hand-sides of a constraint c are respectively denoted by $body(c)$ and $head(c)$. When no confusion is possible, quantifiers are omitted.

A set I of facts satisfies a constraint c, denoted by $I \models c$, if for every homomorphism h from the variables in $body(c)$ into constants in I, the following holds:
– If c is positive: if $h(body(c))$ is in I then there is an extension h' of h such that $h'(head(c))$ is in I.
– If c is negative: if $h(comp(\mathbf{X}'))$ is true in I then depending on c, either $h(L(\mathbf{X}))$ is not in I or one of the two atoms $h(L_1(\mathbf{X}_1))$ or $h(L_2(\mathbf{X}_2))$ is not in I.

Given a set of constraints $\mathbb{C}$, I satisfies $\mathbb{C}$, denoted by $I \models \mathbb{C}$, if for every c in $\mathbb{C}$, $I \models c$ holds.

Example 4.1. Consider the set $\mathbb{C}_a$ with:
c_a : freeTransp(X_{site}, Z) $\rightarrow$ Ecolabel(Z).
c_b : Ecolabel(Z) $\rightarrow$ pCheck(Z, X_o)
which restricts zero-fare public transport to those which have an ecolabel (c_a) and establishes that an ecolabel should have been attested by a pollution checking performed by an official organism (c_b). The database instance D_1 = {freeTransp($Paris, Velib$)} violates c_a while D_2 = {freeTransp($Paris, Velib$), Ecolabel($Velib$), pCheck($Velib, EuropeanCommission$)} satisfies $\mathbb{C}_a$. □

Positive constraints are *linear* LAV (local-as-a-view) TGD (Tuple Generating Dependency) – *i.e.,* each body and head has a unique atom ([4]). Negative constraints are denial dependencies with one or two atoms in their bodies.

The computation of the answers of a query q on an instance D, involves the instantiation of $body(q)$. In our approach, we verify whether the instantiated atoms in $body(q)$ satisfies constraints in $\mathbb{C}$ in order to avoid answers built from facts which violate $\mathbb{C}$. To perform this task, starting with atoms in $body(q)$, we are inspired by the well-known *chase* procedure (see [3] for explanations) to generate new atoms (which are incorporated in the query's body). The chase on a weakly acyclic [20] set of positive constraints is the basis of the query rewriting approach used here and presented in [9]. The following example illustrates this re-writing method.

Example 4.2. Consider again the set $\mathbb{C}_a$ of Example 4.1. Let $Q_a(Y) \leftarrow$ freeTransport($Paris, Y$) be a query on $\mathsf{Ctx}_a = (\mathbb{C}_a, \mathbb{G}_a)$. Starting with atoms in Q_a's body, in a chase-based procedure, atoms Ecolabel(Y) and pCheck(Y, X_o) are produced and added to Q_a's body. The new, rewritten query to be evaluated is $Q_a(Y) \leftarrow$ freeTransport($Paris, Y$), Ecolabel(Y), pCheck(Y, X_o) which ensures that answers are filtered according to Ctx_a. □

5 QUERYING WITHIN A CONTEXT

This paper extends previous work in [9, 10, 14] towards our goal of providing adequate querying capabilities in order to support graph analysis tasks and to ensure data quality. Firstly, the notion of *valid* conjunctive query introduced in [9, 10] is extended to (recursive) datalog programs (Section 5.1) and group-by queries (Section 5.2). Then, in Section 5.3, we adapt the Beta-algebra of [14], which offers an operator capable of representing diverse measurements typical of graph analysis, to our approach. Our resulting querying environment allows graph analytic queries whose results are valid *w.r.t.* a given context.

5.1 Validity

The semantics of a query involving a datalog program is computed by using the immediate consequence operator and thus, at each iteration, only valid data computed on the previous step should be taken into account. In other words, data is filtered at each step of the computation.

Example 5.1. Let us consider the first two rules of Table 1 and query $Q_b(X_{from}, Alesia, X_{time}) \leftarrow$ Connexion$(X_{from}, Alesia, X_{time}, X_{cfp})$ on context Ctx_1. Assume a database instance D_b containing only the facts: Transp(*Montsouris*, *P.Orleans*, *tram*, 1.5, 159), Transp(*P. Orleans*, *JeanMoulin*, *tram*, 1, 106), Transp(*JeanMoulin*, *Didot*, *tram*, 1, 106), Transp(*Didot*, *Alesia*, *bus*, 5, 4528). This instance corresponds to an extract of the graph illustrated in Figure 2. During the computation of Connexion we obtain Connexion(*Montsouris*, *Didot*, 3.5, 371). Although transport from *Didot* to *Alesia* exist in D_b, as $\mathbb{C}_1$ does not allow the use of buses, the connection from *Montsouris* to *Alesia* cannot be computed. □

The following definitions formalize the notion of validity used in this paper.

Definition 5.2 (Valid instantiated literal). Let $\mathsf{Ctx} = (\mathbb{C}, \mathbb{G})$ be a context and D a database instance on $\mathbb{G}$. Let f be an instantiated literal such that $f = L_0(u)$ where L_0 is a predicate in $\mathbb{G}$. The validity of f on D, for context Ctx, is recursively defined as follows (we then say that f is *valid* on D for context Ctx):
Basis step: If predicate L_0 is an extensive predicate in D and if there exists h_1 such that for all $L \in h_1(T^*_{\mathbb{C}_P}(f))$ then the following conditions hold:
(1) there is a positive answer for $q_0() \leftarrow L$ on D;
(2) for each $c \in \mathbb{C}_N$ of the form $(\forall \mathbf{X})((comp(\mathbf{X}') \wedge L_1(\mathbf{X})) \rightarrow \bot)$, there is no homomorphism ν such that $\nu(L_1) = L$ and $\nu(comp(\mathbf{X}')$ is true and
(3) for each $c \in \mathbb{C}_N$ of the form $(\forall \mathbf{X})((comp(\mathbf{X}') \wedge L_1(\mathbf{X}_1) \wedge L_2(\mathbf{X}_2)) \Rightarrow \bot)$, if there is a homomorphism ν such that[1] $\nu(L_i(\mathbf{X}_i)) = L$, then there is no homomorphism ν' that extends ν and for which:
  (i) $\nu'(comp(\mathbf{X}'))$ is true and
  (ii) there is a positive answer for $q'() \leftarrow \nu'(L_{\bar{i}}(\mathbf{X}_{\bar{i}}))$.
Recursive step: If predicate L_0 is an intensive predicate in $\mathbb{P}$ then there exists a rule $r \in \mathbb{P}$ and a homomorphism ν such that $\nu(head(r)) = L_0(u)$ and all atoms in $\nu(body(r))$ are valid. □

Let Valid(f, Ctx, D) be a function which renders *true* when f is valid according to Definition 5.2 Otherwise, Valid(f, Ctx, D) returns $false$.

[1] In our notation, if $i = 1$ then $\bar{i} = 2$ and vice-versa.

Definition 5.3 (Valid answers). Let q be a query which may refer to a query program $\mathbb{P}$. The set of q' s valid answers for context Ctx over a database instance D is defined as follows:

$$valAns\ (q, \mathsf{Ctx}, D) = \{t \mid q(t) \in T^*_{(\mathbb{P}\cup\{q\})}(D) \text{ and } \mathsf{Valid}(q(t), \mathsf{Ctx}, D)\ \}.\ \square$$

Example 5.4. Suppose the instance D_c = { Transp(*Alesia*, *MoutonDuvernet*, *metro*, 1, 106), Transp(*MoutonDuvernet*, *Denfert*, *metro*, 1.5, 133), Transp(*Alesia*, *MoutonDuvernet*, *bus*, 4, 2012), Transp(*MoutonDuvernet*, *Daguerre*, *bus*, 2, 1006), Transp(*Daguerre*, *Denfert*, *bus*, 2, 1006), Type(*metro*, *Rail*), Type(*bus*, *Diesel*) }. Notice that, in this instance, the Transp relation is an extract of the graph illustrated in Figure 2.

Let $Q_c(X) \leftarrow$ Connexion(*Alesia*, X, Y, Z). The fact f = Connexion(*Alesia*, *MoutonDuvernet*, 1, 106) is valid since constraints c_1 and c_2 are verified: D_c contains Type(*metro*, *Rail*) (Definition 5.2, (1)) and the carbon footprint between these two stations is inferior to 500 (Definition 5.2, (2)). Moreover Connexion(*Alesia*, *Denfert*, 2.5, 239) is in the fixpoint $T^*_{\mathbb{C}_1}(f)$ and satisfy conditions in Definition 5.2-Recursive Step. Thus, $\mathsf{Valid}(Q_c\mathsf{Ctx}_1, D_c) = \{MutonDuvernet, Denfert\}$. Note that, in Example 5.1, $\mathsf{Valid}(Q_b, \mathsf{Ctx}_1, D_b) = \emptyset$. $\square$

5.2 Group-by Query

Group-by queries are very important when dealing with data analysis. We adapt the *group by query* q_{ag}, presented in [10], to our purpose. It has the following format:

$$q_{ag}(\mathbf{Y}, a_1, \ldots, a_m) \quad \leftarrow aggr(\phi(\mathbf{X}), \mathbf{Y}, a_1 = f_1(\mathbf{Z}_1), \ldots, \quad a_m = f_m(\mathbf{Z}_m))$$

where $aggr$ is a second order predicate, $\phi(\mathbf{X})$ is a conjunction of atoms over a schema, $\mathbf{Y}$ is the set of grouping attributes, each $\mathbf{Z}_i$ is a set of aggregate attributes, $a_1 \ldots a_m$ are new variables not existing in $\mathbf{X}$ and each f_i is an aggregate function. Moreover, $\mathbf{Y} \subseteq \mathbf{X}$ and for all $1 \leq i \leq m$ we have $\mathbf{Z}_i \subseteq \mathbf{X}$ and $\mathbf{Y} \cap \mathbf{Z}_i = \emptyset$.

Example 5.5. In Table 1, query Q_1 groups Connexion lines that coincide in their origin and destination (group-by function on attributes X_{to} and X_{from}) and for each group computes the average (AVG) over their attributes X_{time} and X_{cfp}. $\square$

We can now formally present the semantics of a group-by query for a given context Ctx.

Definition 5.6 (Group-by query). The set of answers of q_{ag} for context Ctx over D , denoted by $ans(q_{ag}, \mathsf{Ctx}, D)$, is the set of tuples t such that each t is the concatenation of a tuple u and m real values a, *i.e.,* $t = u.a_1.a_2.\ldots.a_m$ (for $m \geq 0$) defined as follows:

• $u \in valAns(q_1, \mathsf{Ctx}, D)$ where q_1 is the conjunctive query $q_1(X_{i_1}, \ldots, X_{i_k}) \leftarrow \phi(X_1, \ldots, X_n)$.

• For each real value a, we have $a = func(valAns(q_2, \mathsf{Ctx}, D))$ where q_2 is the conjunctive query $q_2(X_{j_1}, \ldots, X_{j_{k_l}}) \leftarrow h_u(\phi(X_1, \ldots, X_n))$ and h_u is the homomorphism such that $h_u(X_{i_1}, \ldots, X_{i_k}) = u$. The answer of q_2 is a set of k_l-tuples. Each a value is obtained by the application of a function $func$ on this set of tuples. $\square$

5.3 Beta-Query

In [14] the relational algebra is extended by adding the beta operator. By focusing on data aggregation along the traversal of the relations this new operator allows the definition of different measurements, necessary in graph-based analysis. This paper proposes the β-query, which ensures validity *w.r.t.* a

context, and is a datalog version which extends the operator in [14]. The following example motivates the use of β-queries.

Example 5.7. We are interested in computing minimal carbon footprint to travel from *Alesia* station to any other reachable station in our transportation graph, restrained by Ctx_1. This is a classical problem in graph theory, having a more efficient solution when we avoid computing all paths to later select the minimal ones. In other words, on the basis of Dijkstra's algorithm, the idea is to compute the minimal carbon footprint at each iteration and to abandon paths which are already known as non minimal ones. Let us consider now query Q_β on Ctx_1. A β-query needs some input parameters. First, we define the query that is initially evaluated on our transportation graph as $q_{follows}(X_{from}, X_{to}, X_{cfp}) \leftarrow \mathsf{Transp}(X_{from}, X_{to}, X_{means}\ X_{time}, X_{cfp})$. Answers to this query are filtered according to Ctx_1, *i.e.,* it is evaluated in its rewritten format, namely, $q_{follows}(X_{from}, X_{to}, X_{cfp}) \leftarrow \mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{cfp}), Type(X_{means}, Rail), X_{cfp} <= 500$. The result of $q_{follows}$ is thus the railway version of our transportation graph where every edge respects the carbon limit of 500. Secondly, we define the fact $q_{set}(Alesia, Alesia, 0, 0)$ which designates our start station (a 0-path connexion).

From this input information the β-query associated datalog program is defined. Results obtained during the evaluation of this datalog program are those which will be used to compute the minimal carbon footprint. Following Dijkstra's algorithm the idea is to build, at each iteration of the program, a set of nodes that have minimum (carbon footprint) distance from the source. At each step of the computation, new edges are introduced in paths being constructed, either trying to reach further stations or to optimize carbon footprint for already known stations. Only minimal paths between *Alesia* and a station N are considered in the computation of the next step. □

Roughly speaking, a β-query represents iterative computations on a graph. Specific parts of the computation are represented in the β-query as sub-operations *set, map* and *reduce*. These operations allow users to control how the metrics are calculated as the graph is traversed. The first sub-operation is *set*, which defines the facts with initial values set at the beginning of the computation. *Map* defines a function that controls how the calculated values are spread to neighbors in the graph. *Reduce* aggregates incoming values from the *map* phase into a new metric value for each node. Finally, a fourth sub-operation consists in the revision of the *set* sub-operation. This new *set* defines how new computed values will replace old values from the previous iteration of the β-query.

Now, more precisely, let us show how we build the datalog program associated to a β-query *i.e.,* a group-by query having the format

$$q_\beta(\mathbf{X}, c) \leftarrow aggr(q_{setRes}(\mathbf{X}, c', i), \mathbf{X}, c = f_\beta(c'))$$

where f_β is an *aggregation function* and query q_{setRes} is defined by one of the two rules below, according to the value of the boolean constant PartialRes. Intuitively, PartialRes indicates whether the final result should be computed from all intermediary results or just from the result obtained in the last iteration step.

$$q_{setRes}(\mathbf{X}, c, i) \leftarrow q_{set}(\mathbf{X}, c, i), q_{max}(\mathbf{X}, i), (\textsc{PartialRes} = false)$$
$$q_{setRes}(\mathbf{X}, c, i) \leftarrow q_{set}(\mathbf{X}, c, i), (\textsc{PartialRes} = true)$$

Clearly, q_{setRes} is based on two other queries. The first one is the group-by query q_{max} which groups the resulting values of q_{set} on the set of attributes $\mathbf{X}$ and, for each group, computes the maximum value of i.

$$q_{max}(\mathbf{X}, i) \leftarrow aggr(q_{set}(\mathbf{X}, c, i'), \mathbf{X}, i = max(i'))$$

Answers for query q_{set} result from the evaluation of a recursive Datalog program. We start with $i = 0$ and the evaluation of initial queries q_{set} and $q_{follows}$ for context Ctx on the database instance. Then, each step i of the program evaluation computes two different queries. The first one is q_{map} which, by joining q_{set} and $q_{follows}$ on specific attributes, allows progression on the data graph navigation. This query has the following format:

$$q_{map}(\mathbf{X}, \mathbf{Y}, \mathbf{c}, i) \leftarrow \quad q_{set}(\mathbf{X}, \mathbf{c}_1, i_0),\ q_{follows}(\mathbf{Z}, \mathbf{Y}, \mathbf{c}_2),\ OK(i),\ \ \mathbf{c} = \mathbf{f}_{map}(\mathbf{c}_1, \mathbf{c}_2),\ i = i_0 + 1$$

where $\mathbf{Z}$ is a subset of $\mathbf{X}$, $OK(i) \leftarrow i \leq N$; and $\mathbf{f}_{map}$ is a function (usually an arithmetic function) applied on counters $\mathbf{c}_1$, and $\mathbf{c}_2$ (*e.g.* multiplication, division, etc). The constant N imposes a limit on the number of iterations. It might be replaced by other converging test mechanism.

Then, query q_{reduce} groups intermediate results computed by q_{map}:

$$q_{reduce}(\mathbf{Z}, \mathbf{c}, i) \leftarrow \quad aggr(q_{map}(\mathbf{X}, \mathbf{Y}, \mathbf{c}_1, i), \mathbf{Z} \cup \{i\}, \mathbf{c} = \mathbf{f}_{red}(\mathbf{c}_1))$$

where $\mathbf{Z}$ is a subset of $\mathbf{X} \cup \mathbf{Y}$ such that $\mid \mathbf{Z} \mid = \mid \mathbf{X} \mid$ and $\mathbf{f}_{red}$ is an aggregation function.

Finally, the join of q_{set} values (those computed in a previous step i') with values computed by q_{reduce} (at the current step i) allows us to obtain the new values for q_{set} (q_{set} values at step i).

$$q_{set}(\mathbf{X}, \mathbf{c}, i) \leftarrow \quad q_{reduce}(\mathbf{X}, \mathbf{c}, i), q_{set}(\mathbf{X}, \mathbf{c}', i'), cond(\mathbf{c}, \mathbf{c}')$$
$$q_{set}(\mathbf{X}, \mathbf{c}, i) \leftarrow \quad q_{reduce}(\mathbf{X}, \mathbf{c}, i), \neg q_{set}(\mathbf{X}, \mathbf{c}', i')$$

It is interesting to note that the answer set for the first q_{set} query above is composed by tuples in the intersection of q_{set} and q_{reduce} and respecting a given condition (*cond*) involving counters **c** and **c**', while for the second query, it is composed by tuples appearing exclusively in the latter computation of q_{reduce}.

We continue the discussion started in Example 5.7 and present the datalog program associated to Q_β (Table 1).

Example 5.8. In the case of Example 5.7, PartialRes is *true* because the minimal carbon footprint between *Alesia* and a station N can be obtained at any step (not necessarily at the last one).

Now, the map operation is expressed by the following rule.

$$q_{map}\ (X_{from}, X_{by}, X_{to}, X_{cfp}, i) \leftarrow \quad q_{set}(X_{from}, X_{by}, X^1_{cfp}, i^1),\ q_{follows}(X_{by}, X_{to}, X^2_{cfp}),$$
$$X_{cfp} = X^1_{cfp} + X^2_{cfp},\quad i = i^1 + 1$$

Notice that i represents the number of links (edges on the graph) used to go from $X_{from} = Alesia$ to X_{to}. Moreover, the join between q_{set} and $q_{follows}$ relations is done on the second attribute of q_{set} and the first of $q_{follows}$. This ensures the construction of paths on the railway version of our transportation graph. Here, the $\mathbf{f}_{map}$ computes the carbon footprint of a new path by adding the intermediary carbon footprints. Notice also that $Ok(i)$ is always $True$, since the computation continues until reaching a fixpoint.

To illustrate the computation, in this example, we consider part of the transport graph, concerning connexions from *Alesia* to *Cite-Universitaire* (Figure 2). To avoid overcharging our figures, we take connexions in only one direction. In this context, Figure 3 shows three steps of our computation. As already explained in Example 5.7, at step 0, query q_{set} indicates the start station and $q_{follows}$ returns all graph edges respecting constraints in Ctx_1 (railway edges respecting cfp limit). The idea of q_{map} is to increase paths' lengths, going further, adding one more station to previous paths. Here, at

STEP 0
$q_{set}(Alesia, Alesia, 0, 0)$
$q_{follows}(Alesia, P.Orleans, 160)$
$q_{follows}(Alesia, MoutonDuvernet, 106)$
$q_{follows}(MoutonDuvernet, Denfert, 133)$
$q_{follows}(Denfert, CiteUniv, 500)$
$q_{follows}(P.Orleans, Montsouris, 159)$
$q_{follows}(Montsouris, CiteUniv, 106)$

STEP 1
$q_{map}(Alesia, Alesia, P.Orleans, 160, 1)$
$q_{map}(Alesia, Alesia, MoutonDuvernet, 106, 1)$
$q_{reduce}(Alesia, P.Orleans, 160, 1)$
$q_{reduce}(Alesia, MoutonDuvernet, 106, 1)$
$q_{set}(Alesia, Alesia, 0, 0)$
$q_{set}(Alesia, P.Orleans, 160, 1)$
$q_{set}(Alesia, MoutonDuvernet, 106, 1)$

STEP 2
$q_{map}(Alesia, Alesia, P.Orleans, 160, 1)$
$q_{map}(Alesia, Alesia, MoutonDuvernet, 106, 1)$
$q_{map}(Alesia, P.Orleans, Montsouris, 319, 2)$
$q_{map}(Alesia, MoutonDuvernet, Denfert, 239, 2)$
$q_{reduce}(Alesia, P.Orleans, 160, 1)$
$q_{reduce}(Alesia, MoutonDuvernet, 106, 1)$
$q_{reduce}(Alesia, Montsouris, 319, 2)$
$q_{reduce}(Alesia, Denfert, 239, 2)$
$q_{set}(Alesia, Alesia, 0, 0)$
$q_{set}(Alesia, P.Orleans, 160, 1)$
$q_{set}(Alesia, MoutonDuvernet, 106, 1)$
$q_{set}(Alesia, Montsouris, 319, 2)$
$q_{set}(Alesia, Denfert, 239, 2)$

STEP 3
$q_{map}(Alesia, Alesia, P.Orleans, 160, 1)$
$q_{map}(Alesia, Alesia, MoutonDuvernet, 106, 1)$
$q_{map}(Alesia, P.Orleans, Montsouris, 319, 2)$
$q_{map}(Alesia, MoutonDuvernet, Denfert, 239, 2)$
$q_{map}(Alesia, Montsouris, CiteUniv, 425, 3)$
$q_{map}(Alesia, Denfert, CiteUniv, 739, 3)$
$q_{reduce}(Alesia, P.Orleans, 160, 1)$
$q_{reduce}(Alesia, MoutonDuvernet, 106, 1)$
$q_{reduce}(Alesia, Montsouris, 319, 2)$
$q_{reduce}(Alesia, Denfert, 239, 2)$
$q_{reduce}(Alesia, CiteUniv, 425, 3)$
$q_{set}(Alesia, Alesia, 0, 0)$
$q_{set}(Alesia, P.Orleans, 160, 1)$
$q_{set}(Alesia, MoutonDuvernet, 106, 1)$
$q_{set}(Alesia, Montsouris, 319, 2)$
$q_{set}(Alesia, Denfert, 239, 2)$
$q_{set}(Alesia, CiteUniv, 425, 3)$

Fig. 3. Computation of minimal cfp - starting at Alesia

STEP 1 of Figure 3, q_{map} returns all paths which are 1-station away from Alesia, namely, *P.Orleans* and *MountonDuvernet*.

Next, for q_{reduce}, we have:

$$q_{reduce}\ (X_{from}, X_{to}, X_{cfp}, i) \leftarrow aggr(q_{map}(X_{from}, X_{inter}, X_{to}, X^1_{cfp}, i),\ \ X_{from}, X_{to}, i,\ \ X_{cfp} = MIN(X^1_{cfp}))$$

where results are grouped on attributes X_{from}, X_{to}, i and the aggregation function MIN is used on carbon footprint values. Therefore, relation q_{reduce} has the minimal carbon footprint for each path (from *Alesia* to a reachable node by rail) on each iteration i.

In Figure 3, we notice that on STEP 1, q_{reduce} returns similar results to those obtained by q_{map} since there are no different ways to go from *Alesia* to *P.Orleans* or *MountonDuvernet*.

Finally, q_{set} is redefined by the following two rules:

$$q_{set}\ (X_{from}, X_{to}, X_{cfp}, i) \leftarrow q_{reduce}(X_{from}, X_{to}, X_{cfp}, i), q_{set}(X_{from}, X_{to}, X^1_{cfp}, i^1),\ \ X_{cfp} < X^1_{cfp} \qquad (*)$$

$$q_{set}(X_{from}, X_{to}, X_{cfp}, i) \leftarrow q_{reduce}(X_{from}, X_{to}, X_{cfp}, i), \neg q_{set}(X_{from}, X_{to}, X^1_{cfp}, i^1) \qquad (**)$$

The first one updates the carbon footprint for two-node connexions already computed (in previous steps). Updates are performed only when the new computed value corresponds to a path proposing less energy consumption. The second rule deals with new nodes we can reach from X_{from} =*Alesia*.

After updating q_{set} the computation continues until the fixpoint. In the example of Figure 3, we notice that after STEP 3 no other changes are possible. Notice that, in all steps illustrated in that figure, only the second rule ($**$) is used to update q_{set}. Indeed, in the transport graph extract used in our example, there are no paths from *Alesia* to another station N having different number of intermediate stations (refer to $q_{follows}$ on STEP 0). Thus, at each step of the computation, only new paths are added to q_{set}. Notice however that there are two ways to go from *Alesia* to $CiteUniv$ passing by two intermediate stations: indeed, we have $q_{map}(Alesia, Montsouris, CiteUniv, 425, 3)$ and $q_{map}(Alesia, Denfert, CiteUniv, 739, 3)$ on STEP 3 of Figure 3. But, on this same step, q_{reduce} (Figure 3) selects only the path with minimal cfp between these two stations.

All the above rules are part of the datalog program whose fixpoint gives connections from *Alesia* to a node N reachable by rail and its associated carbon footprint. Each carbon footprint is a minimal value found in a step i of the computation. However, as the final result, we have to find the minimal ones, independently of the iteration number i. The following query computes this final result.

$q_{setRes}(X_{from}, X_{to}, X_{cfp}) \leftarrow aggr(q_{set}(X_{from}, X_{to}, X^1_{cfp}, i), X_{from}, X_{to}, \ \ X_{cfp} = MIN(X^1_{cfp}).$ □

Our querying system intends to be an important tool for platforms focusing on the analysis of networks. Such a platform may then propose graphical user interfaces to help an *end user* to choose or build his context and to execute pre-defined queries. For example, consider a pre-defined query for computing minimal paths. In this case, Q_β (Table 1) can be used: the user just have to specify starting node (X_{from}) or the end node (X_{to}) and the parameter on which minimality is required (X_{cfp} or X_{time}).

A more flexible viewpoint is reserved to *expert users*, capable of customizing a β-query. In such case we can propose the automatic generation of the datalog program associated to a β-query on the basis of some input parameters.

In the next section we present a use case to illustrate graph analysis, via a β-query, on the urban context.

6 GRAPH ANALYSIS IN CITIES: AN USE CASE

This section presents a simplified use case where Paris-city government aims to analyse the best place to install a new *crèche*. Simplifications (*w.r.t.* the number of available predicates and constraints) allow us to focus on how the β-query is used. Therefore, we consider the constraints presented in Table 3. Context Ctx_2 establishes that we are only interested in railways (c_1), that we consider places (X_{from}) near a garden (c_3) and a station (c_4), which should not be near an existing crèche (c_5).

Context 2: Set of constraints $\mathbb{C}_2$ of context Ctx_2

c_1	$\mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{km}, X_{cfp})$	$\rightarrow$	$\mathsf{Type}(X_{means}, Rail)$.
c_3	$\mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{km}, X_{cfp})$	$\rightarrow$	$\mathsf{Environment}(X_{from}, Garden)$.
c_4	$\mathsf{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{km}, X_{cfp})$	$\rightarrow$	$\mathsf{Environment}(X_{from}, Station)$.
c_5	$\mathsf{Environment}(X_{from}, Station), \mathsf{Environment}(X_{from}, Creche)$	$\rightarrow$	$\bot$.

Table 3. Context $\mathsf{Ctx}_2 = (\mathbb{C}_2, \mathbb{G})$ on a transportation database.

$NeighbourCount(X_{from}, c) \leftarrow$	$agg(\text{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{km}, X_{cfp}), X_{from}, c = COUNT(X_{to}))$
$q_{follows}(X_{from}, X_{to}, c) \leftarrow$	$\text{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{km}, X_{cfp}), NeighbourCount(X_{from}, c).$
$q_{set}(X_{from}, r, i) \leftarrow$	$\text{Transp}(X_{from}, X_{to}, X_{means}, X_{time}, X_{km}, X_{cfp}), r = 100, i = 0.$
$q_{map}(X_{from}, X_{to}, r, i) \leftarrow$	$q_{set}(X_{from}, r_0, i), q_{follows}(X_{from}, X_{to}, c),\ r = r_0/c,\ i = i_0 + 1, i < 10.$
$q_{reduce}(X_{to}, r, i) \leftarrow$	$agg(q_{map}(X_{from}, X_{to}, r_0, i),\ \ X_{to}, i,\ \ r = SUM(r_0)).$
$q_{set}(X_{from}, r, i) \leftarrow$	$q_{reduce}(X_{from}, r, i).$
$q_{max}(X_{from}, i) \leftarrow$	$aggr(q_{set}(X_{from}, r, i'), X_{from}, i = MAX(i'))$
$q_{SetRes}(X_{from}, r) \leftarrow$	$q_{set}(X_{from}, r, i), q_{max}(X_{from}, i)$

Table 4. Datalog program associated to the β-query computing the rank of transportation graph nodes.

	1	2	3	4	5	6	7	8	9	10	19	20	29	30	39	40	49	50
Alesia	100	75	125	75	125	75	125	75	125	75	125	75	125	75	125	75	125	75
CiteUniversitaire	100	100	88	113	81	119	78	122	77	123	75	125	75	125	75	125	75	125
DenfertRochereau	100	100	100	88	113	81	119	78	122	77	125	75	125	75	125	75	125	75
JeanMoulin	100	25	75	31	69	34	66	36	64	37	62	37	63	37	63	37	63	37
MairieMontRouge	100	25	75	31	69	34	66	36	64	37	62	37	63	37	63	37	63	37
MontSouris	100	75	125	75	125	75	125	75	125	75	125	75	125	75	125	75	125	75
MoutonDuvernet	100	100	88	113	81	119	78	122	77	123	75	125	75	125	75	125	75	125
PortedOrleans	100	300	125	275	138	263	144	256	147	253	150	250	150	250	150	250	150	250

Table 5. Rank computation: the table shows for each station respecting Ctx_2 its rank on step i.

To find the best place for a crèche one should also take into account transport possibilities. The goal here is to choose a place easily reachable via the available transport network. We use the following β-query:

$$Q_{\beta_1}(X_{from}, X_{rank}) \leftarrow q_{SetRes}(X_{from}, X_{rank})$$

to compute the *rank* (X_{rank}) of a place (X_{from}) *w.r.t.* Ctx_2. Indeed, the above query is associated to the datalog program presented in Table 4 which implements a PageRank-like algorithm. We recall that PageRank is a link analysis algorithm which assigns a numerical weighting to nodes in a graph, with the purpose of "measuring" its relative importance within the graph. That is, in our example, the algorithm can measure the attractiveness of a station *w.r.t.* possible connexions with other stations on the transportation graph. However, this attractiveness is bounded by restrictions imposed in Ctx_2. In fact, the new crèche should be placed not very far from a railway station (c_1 and c_4) and a garden (c_3), the latter allowing occasional walks with children. Moreover, constraint c_4 ensures the non existence of a crèche as a facility in the selected station environment - a way to avoid placing the new crèche in a environment already having another crèche.

To run our use case as a small example, we have assumed only the graph of Figure 2 where we arbitrarily placed existing crèches and gardens.

Table 5 shows the rank computation, *w.r.t.* Ctx_2, for each station depicted in Figure 2. Although in this example the rank presents a 1-step oscillation (*e.g. Alesia* has rank 75 on step 6, rank 125 on step 7 and comes back to rank 75 on step 8), we can draw some conclusions concerning the best place for a new crèche. Notice that convergence is reached when we consider just even (or odd) steps. For instance, the rank of *Alesia* stabilizes on 125 when only odd steps are considered. When we increase the number of connexions, stability can be reached. Anyway, from the computed values, it is possible to conclude that *Porte d'Orléans* is the most attractive station near which a new crèche should be placed in accordance to Ctx_2.

The above results have been obtained by our in-memory prototype, being implemented in Java.

7 RELATED WORK

Different graph databases and languages are currently available [5, 19, 27]. Although the options being offered have advantages in terms of data storage and management, they lack query capabilities to capture network properties typical of graph analysis and means to define the user's context through constraints.

Graph query languages are often based on conjunctive regular path queries (CRPQs). CRPQs are the basis for several graph languages, such as GraphLog [11] and SPARQL[2]. Recent developments have extended CRPQs in order to allow constraints over path properties. These types of queries have been described as extended conjunctive regular path queries (ECRPQs) [6]. ECRPQs also allow paths to be returned as query results. These queries are all focused on data selection and support only simple cases of analysis.

Lately there has been renewed interest in expressive query languages to tackle modern analysis problems. To overcome the limits of traditional query languages, Datalog has been proposed as the basis of the new languages [13], and extensions to support aggregates in recursive logic rules have been presented as in [17]. Prominent projects in this area exist. SociaLite [22] offers integration between Datalog and procedural languages, allowing embedded Datalog formulas inside loops and conditionals. EmptyHeaded [2] is a relational engine for graph processing that offers a Datalog-like language with native support for fixpoint queries. BigDatalog [23] extends Spark with recursion. Datalography [18] proposes an evaluation engine for executing graph analysis over BSP-style graph processing engine. These projects focus on optimization of the parallel execution of queries through the use of data structures that yield better execution plans of multi-way joins.

The work presented in this paper, in contrast, revisits query expressiveness and usability, offering higher-level constructs for users to specify the query context (constraints) and parameters of the graph analysis (β-queries). Although we do not focus on query optimization, the mentioned approaches are compatible with our querying strategies and are currently being considered in our implementations.

Constraints are taken into account in the context of RDF technologies such as ShEx [25], SPIN [15], and SHACL [16]. However, their focus is on schema and ours is on integrity constraints. Stardog [26] deals with constraints which are closer to ours. The mentioned works, however, do not emphasize query expressiveness in the context of graph analysis.

8 CONCLUDING REMARKS

Smart cities face several challenges that require processing of a large volume of data provided by different sources, available for different goals. In such environment, the ability of performing context-driven analysis on graphs is essential to the decision process. This paper contributes in this direction, proposing: (i) a declarative query language offering tools for graph analysis in a context-driven environment which allows reasoning with confidence over multistore systems [7] and (ii) a querying system which filters answers obtained from a database (distributed or not) according to a given context. It adopts the philosophy that data quality is "everyone's business" [24] placing quality requirements at

[2]http://www.w3.org/TR/sparql11-query

user's hand. Data in sources may be inconsistent *w.r.t.* context restrictions, but only consistent subsets are returned to the user.

Indeed, our contribution is the result of the integration of constraint checking (which enables context-aware queries) and query expressiveness for graph analysis (which enables recursive computations over networks). To the best of our knowledge, this is the first querying environment covering these two capabilities. To better exploit our system's possibilities, we are currently working on its construction on the top of a think-like-a-vertex parallel graph processing platform: the one proposed in [21] and Giraph (similar to what is done in [18]).

Acknowledgements: This work was partially supported by APR-IA Girafon and PEPS-Multipoint.

REFERENCES

[1] [n. d.]. Eco Comparateur. At http://quizz.ademe.fr/eco-deplacements/comparateur/. ([n. d.]).

[2] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.* 42, 4 (2017), 20:1–20:44.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[4] Foto N. Afrati and Phokion G. Kolaitis. 2009. Repair checking in inconsistent databases: algorithms and complexity. In *Database Theory - ICDT, 12th International Conference, Russia, Proceedings*. 31–41.

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

[6] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst* 37, 4 (2012), 31.

[7] Carlyna Bondiombouy and Patrick Valduriez. 2016. Query processing in multistore systems: an overview. *IJCC* 5, 4 (2016), 309–346.

[8] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30, 1-7 (1998), 107–117.

[9] Jacques Chabin, Mírian Halfeld Ferrari, Béatrice Markhoff, and Thanh Binh Nguyen. 2018. Validating Data from Semantic Web Providers. In *SOFSEM : Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, J Proceedings*. 682–695.

[10] Jacques Chabin, Mirian Halfeld-Ferrari, and Thanh Binh Nguyen. 2016. *Querying Semantic Graph Databases in View of Constraints and Provenance*. Technical Report. LIFO- Université d'Orléans, RR-2016-02.

[11] Mariano Consens and Alberto O. Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *In Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 404–416.

[12] Laurent D'Orazio, Mirian Halfeld-Ferrari, Carmem S. Hara, Nadia P. Kozievitch, and Martin A. Musicante. 2017. Graph Constraints in Urban Computing:Dealing with conditions in processing urban data. In *DARLI-AP, International workshop on Data Analytics solutions for Real-LIfe APplications in conjunction with The 3rd IEEE International Conference on Smart Data, England, UK.*

[13] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. 2012. LogicBlox, Platform and Language: A Tutorial. In *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria. Proceedings*. 1–8.

[14] Luiz Gomes Jr., Bernd Amann, and André Santanchè. 2015. Beta-Algebra: Towards a Relational Algebra for Graph Analysis. In *Proceedings of the Workshops of the EDBT/ICDT Joint Conference, Brussels, Belgium*. 157–162.

[15] H. Knublauch, J. A. Hendler, and K. Idehen. 2011. SPIN - Overview and Motivation. W3C Member Submission. http://www.w3.org/Submission/2011/SUBM-spin-overview-20110222. (2011).

[16] H. Knublauch and A. Ryman. 2017. Shapes Constraint Language (SHACL). W3C First Public Working Draft, W3C. http://www.w3.org/TR/2015/WD-shacl-20151008/.. (2017).

[17] Mirjana Mazuran, Edoardo Serra, and Carlo Zaniolo. 2013. Extending the power of datalog recursion. *VLDB J.* 22, 4 (2013), 471–493.

[18] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *IEEE International Conference on Big Data, BigData, Washington DC, USA*. 56–65.

[19] Inc Neo4j. 2017. Introduction to Cypher. https://neo4j.com/developer/cypher-query-language/. (2017).

[20] Adrian Onet. 2013. The Chase Procedure and its Applications in Data Exchange. In *Data Exchange, Integration, and Streams*. 1–37.

[21] Raqueline R. M. Penteado, Rebeca Schroeder, and Carmem S. Hara. 2016. Exploring Controlled RDF Distribution. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2016, Luxembourg*. 160–167.

[22] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2015. SociaLite: An Efficient Graph Query Language Based on Datalog. *IEEE Trans. Knowl. Data Eng* 27, 7 (2015), 1824–1837.

[23] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the International Conference on Management of Data, SIGMOD Conference, San Francisco, CA, USA*. 1135–1149.

[24] Eric Simon. [n. d.]. Agile data management challenges in enterprise big data landscapes. Lecture in XLDB'2017. ([n. d.]). https://indico.in2p3.fr/event/14490/timetable/#20171010.detailed

[25] H. Solbrig and E. Prud'hommeaux. 2014. Shape Expressions 1.0 Definition. W3C Member Submission. http://www.w3.org/Submission/2014/SUBM-shex-defn-20140602. (2014).

[26] Stardog5. 2017. Enterprise Knowledge Graph. http://www.stardog.com/docs/. (2017).

[27] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.