An Extension of Datalog for Graph Queries*

Mirjana Mazuran[1], Edoardo Serra[2], and Carlo Zaniolo[3]

[1] Politecnico di Milano DEI – mazuran@elet.polimi.it
[2] University of Calabria DEIS – eserra@deis.unical.it
[3] University of California, Los Angeles – zaniolo@cs.ucla.edu

Abstract. Supporting aggregates in recursive logic rules is a crucial long-standing problem for Datalog. To solve this problem, we propose Datalog^{FS} that supports queries and reasoning on the number of distinct occurrences satisfying given goals, or conjunction of goals, in rules. By using a generalized notion of multiplicity called frequency, we show that graph queries can be easily expressed in Datalog^{FS}. This simple extension preserves all the desirable semantic and computational properties of logic-based languages, while significantly extending their application range to support efficiently page-rank, and social-network queries.

1 Introduction

Due to the emergence of many important application areas we are now experiencing a major resurgence of interest in Datalog for parallel and distributed programming [1] and for expressing and supporting subsets of Description Logic for ontological queries [2]. Other lines of work are exploring execution of recursive queries in the MapReduce framework [3] and in Data Stream Management Systems [4]. The abundance of new applications underscores the need to tackle and solve crucial Datalog problems that remain unsolved and restrict its effectiveness by e.g., disallowing the use of aggregates in recursion. This problem is very challenging since basic aggregates violate the requirement of monotonicity on which the least fixpoint semantics of Datalog is based.

Related Work The notion of *stratification* with respect to negation and aggregates is simple for users to master [5, 6]. Unfortunately, stratification (into a finite number of strata) is too restrictive and cannot support the efficient formulation of many graph optimization algorithms, which typically require the use of extrema and counting in recursion [7].

The importance of optimization and graph applications have motivated much research work seeking to solve these problems. These proposals follow three main approaches: *i*) supporting infinite levels of stratifications using Datalog$_{1S}$ programs [6]; *ii*) attempting to preserve the fixpoint computation via continuous aggregates and non-deterministic *choice* constructs [8, 9], and *iii*) seeking to achieve monotonicity by using partial orders that are more general than set-containment [10]. These past solutions had limited generality and often required

* Extended Abstract

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

sophisticated users and compilers. We next introduce Datalog^{FS} which does not suffer from these problems.

In the next section we introduce Datalog^{FS} via simple examples. In Section 3 we introduce constructs that support facts and predicates having multiple occurrences and, in Section 4, we review important graph applications. In Section 5, we show how to implement Datalog^{FS} efficiently.

2 Datalog^{FS} by Example

Consider, for instance a database of facts as follows:
$\mathtt{person(adam).}$ $\mathtt{person(marc).}$ $\mathtt{person(jerry).}$ $\mathtt{person(tom).}$
$\mathtt{son(marc,tom).}$ $\mathtt{son(marc,jerry).}$ $\mathtt{son(tom,eddy).}$ $\mathtt{son(tom,adam).}$ $\mathtt{son(tom,john).}$

The following rule defines fathers with at least two sons:

$$\mathtt{twosons(X)} \leftarrow \mathtt{person(X), son(X,Y1), son(X,Y2), Y2} \neq \mathtt{Y1.}$$

Datalog^{FS} allows the following equivalent expression for our `twosons` rule:

$$\mathtt{twosons(X)} \leftarrow \mathtt{person(X),\ 2{:}[son(X,Y)].}$$

The goal, `I:[b-expression]`, is called a frequency support goal (or *FS-Goal*), and "`I`" is a positive integer, called *Running-FS clause*. The expression in the bracket is called *b-expression*, and can either consists of a single positive predicate or a conjunction of positive predicates [11]. The convenience of FS-goals is clear if we want to find people with many sons:

$$\mathtt{sixsons(X)} \leftarrow \mathtt{person(X),\ 6{:}[son(X,Y)].}$$

will retrieve all persons who have at least six sons. An equivalent rule can be expressed using the $\neq$ operator. Indeed we can start as follows:

$$\mathtt{sixsons(X)} \leftarrow \mathtt{person(X), son(X,Y1), 5{:}[son(X,Y2), Y2} \neq \mathtt{Y1].}$$

and proceed inductively, and obtain a rule containing six goals $\mathtt{son(X,Y_j)}$, where $\mathtt{j} = \mathtt{1}, \ldots, \mathtt{6}$ and $\mathtt{6}\times\mathtt{5}$ goals saying, that every `Y` be different from every other `Y`. If we are interested in links between web pages, which could easily be thousands, it becomes clear that the approach based on $\neq$ is totally impractical, and without FS-goals we would need a COUNT aggregate. Yet, aggregates bring in the curse of non-monotonicity and recursion becomes a problem. At the semantic level, our Datalog^{FS} rules can instead be viewed as standard Horn clauses whereby the standard monotonicity-based semantics of negation-free Datalog is preserved.

We now clarify the scope of variables in Datalog^{FS}. Predicate `friend(X,Y)` denotes that person `X` views person `Y` as a friend (no assumption of symmetry):

Example 1. Pairs of friends (F1, F2) where F1 and F2 have at least 3 friends:

$$\mathtt{popularpair(X,Y)} \leftarrow \mathtt{friend(X,Y), 3{:}[friend(X,V1)], 3{:}[friend(Y,V2)].}$$

Example 2. Pairs of friends (`F1,F2`) who have at least three friends in common

$$\mathtt{sharethree(X,Y)} \leftarrow \mathtt{friend(X,Y), 3{:}[friend(X,V), friend(Y,V)].}$$

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

There are two kinds of variables in rules with FS-goals. The first are those, such as `X` and `Y` in Example 2, that appear in the head of the rule or in some goal outside the b-expression. These will be called *global* variables. They are basically the universally qualified variables of the standard Horn Clauses, and have the whole rule as scope. Variables `X` and `Y` in Example 1 are global for that rule.

Other variables only appear in b-expressions and their scope is *local* to the b-expression, where they appear (e.g. `V1` and `V2` in Example 1, and `V` in Example 2). Thus, $K[\ldots]$ can be viewed as an existential declaration of local variables under the following constraint: there exist at least K assignments of the local variables that satisfy the b-expression. Example 2 states that there exist at least 3 distinct `V` occurrences each denoting a person who is a friend to both `X` and `Y`. The scope of existential variables is local to the b-expression: in Example 1 replacing `V1` and `V2` with `V` would not change the meaning of our rule. Let us now express that an assistant professor to be advanced to associate professor should have an H-index of at least 13:

Example 3. Our candidate must have authored at least 13 papers each of which has been referenced at least 13 times. The database table $\mathtt{author}(\mathtt{Author}, \mathtt{Pno})$ lists all papers (co-)authored by a person, while the atom $\mathtt{refer}(\mathtt{PnFrom}, \mathtt{PnTo})$ denotes that paper `PnFrom` contains a reference to paper `PnTo`.

$$\mathtt{atleast13}(\mathtt{PnTo}) \leftarrow\ \mathtt{13}{:}[\mathtt{refer}(\mathtt{PnFrom}, \mathtt{PnTo})].$$
$$\mathtt{hindex13}(\mathtt{Author}) \leftarrow \mathtt{13}{:}[\mathtt{author}(\mathtt{Author}, \mathtt{Pno}), \mathtt{atleast13}(\mathtt{Pno})].$$

These simple examples could also be expressed using the count aggregate. Yet, count and other aggregates are non-monotonic with respect to the partial ordering defined by set containment, and cannot be used in recursive rules. Indeed, the meaning and efficient implementation of Datalog programs with recursive rules are based on their least fixpoint semantics[4], which is only guaranteed to exist when the program rules define mononotonic mappings. Now, continuous count is obviously monotonic with respect to set-containment. This is formally proven in [11] by rewriting the running FS-construct with equivalent, although inefficient, Horn clauses (that use list and thus we will avoid in the actual implementation, as shown in Section 5).

Final-FS construct and Stratification. The semantics of Datalog^{FS} [11] allows to use variables rather than constants in the specification of FS goals. This is useful, for instance, to find the actual number of sons a person has:

Example 4. How many sons does a person have?

$$\mathtt{csons}(\mathtt{PName}, \mathtt{N}) \leftarrow\ \mathtt{person}(\mathtt{PName}), \mathtt{N}{:}[\mathtt{son}(\mathtt{PName}, \mathtt{Sname})], \neg\mathtt{morethan}(\mathtt{PName}, \mathtt{N}).$$
$$\mathtt{morethan}(\mathtt{PName}, \mathtt{N}) \leftarrow \mathtt{N1}{:}[\mathtt{son}(\mathtt{PName}, \_)], \mathtt{N1} > \mathtt{N}.$$

Thus `csons` must belong to a stratum that is strictly higher than `son`, whereas with respect to `person` it could be in the same stratum or in the one above it. The need to find the maximum value satisfying a running-FS clause is so common that we provide a construct called Final-FS and denoted by the operator `=!`.

[4] Naturally, by "least fixpoint" of a program, we mean "least fixpoint of its immediate consequence operator" [12].

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

Example 5. How many sons does a person have?

$$\mathtt{csons(Name, N)} \leftarrow \mathtt{person(Name), N =![son(Name, \_)]}.$$

The formal semantics of the *Final-FS* construct is defined as the rewriting of Example 5 into Example 4, which makes use of negation, whereby we will require that our Datalog^{FS} programs be stratified w.r.t. Final-FS goals.

Recursive Datalog^{FS}. Consider the following example:
Example 6. Some people will come to the party for sure. Others will also come once they learn that three or more of their friends will come.

$$\mathtt{willcome(X)} \leftarrow \mathtt{sure(X)}.$$
$$\mathtt{willcome(Y)} \leftarrow \mathtt{3{:}[friend(Y, X), willcome(X)]}.$$

One person might be more timid than another, and different people could require a different number of friends before they also join the party. Thus, if `requires(Person, PNumber)` denotes the number of friends required by a person, where `PNumber` must be a positive integer (whereas `sure` denotes people who will come even if none of their friends will), we have the following program:

Example 7. A person will join the party if a sufficient number of friends join.

$$\mathtt{join(X)} \leftarrow \mathtt{sure(X)}.$$
$$\mathtt{join(Y)} \leftarrow \mathtt{requires(Y, K), K{:}[friend(Y, X), join(X)]}.$$

3 Multi-Occuring Predicates

As discussed in [10], there are numerous examples where it is desirable that certain predicates are counted as providing a support level greater than one. For instance, we might use the following representation to denote that the paper with DBLP identifier "MousaviZ11" is currently cited in six papers: `ref("MousaviZ11"):6`. Thus, `Pno = "MousaviZ11"` contributes with count six to the b-expression of the rule:

Example 8. Total reference count for an author.

$$\mathtt{tref(Authr){:}N} \leftarrow \mathtt{N{:}[author(Authr, Pno), ref(Pno)]}.$$

The clauses `":6" and ":N"` used in the above fact and rule head will be called *FS-Assert* clauses. The semantics of programs P with frequency assert clauses is defined by *expanding* it into its $\bar{P}$ equivalent, which is obtained as follows: Each rule in P with head `q(X1, ..., Xn):K` $\leftarrow$ `Body` is replaced by

$$\bar{\mathtt{q}}\mathtt{(X1, \ldots, Xn, J)} \leftarrow \mathtt{lessthan(J, K), Body}.$$

where `lessthan(J, K)` is a recursive predicate that generates all positive integers up to `K`, included. Thus, we have that, as a result of this expansion, `ref("MousaviZ11"):6` contributes with six to the reference count of each author of that paper. A property of frequency statements is that, when multiple statements hold for the same fact *only the largest value* is significant, the others are subsumed and can be ignored.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

Bill-of-materials (BOM) applications represent a well-known example of the need for recursive queries. Our database might contain records $\texttt{assbl}(\texttt{Part}, \texttt{Subpart}, \texttt{Qty})$ which, for each part number, gives the immediate subparts used in its assembly and their quantity (e.g. a bicycle has 1 frame and 2 wheels as immediate subparts). At the bottom of BOM DAG, we find the basic parts that are purchased from external suppliers and described by $\texttt{basic}(\texttt{Pno}, \texttt{Days})$ denoting the days needed to obtain that basic part. Several interesting BOM applications are naturally expressed by combining aggregates and recursion, as follows:

Example 9. How many basic parts does an assembled part contain?

$$\texttt{cassb}(\texttt{Part}, \texttt{Sub}):\texttt{Qty} \leftarrow \texttt{assbl}(\texttt{Part}, \texttt{Sub}, \texttt{Qty}).$$

$$\begin{aligned}
&\texttt{cbasic}(\texttt{Pno}):\texttt{1} \leftarrow && \texttt{basic}(\texttt{Pno}, _).\\
&\texttt{cbasic}(\texttt{Part}):\texttt{K} \leftarrow && \texttt{K}:[\texttt{cassb}(\texttt{Part}, \texttt{Sub}), \texttt{cbasic}(\texttt{Sub})].\\
&\texttt{cntbasic}(\texttt{Prt}, \texttt{C}) \leftarrow && \texttt{C} =![\texttt{cbasic}(\texttt{Prt})].
\end{aligned}$$

The count of basic parts is not retrieved using the goal $\texttt{N}:[\texttt{cbasic}(\texttt{frame})]$ since this will return all positive integers up to the max value. Goal $\texttt{N} =![\texttt{cbasic}(\texttt{frame})]$ is used instead as this returns the exact count of the basic subparts.

Example 10. How many days until delivery?

$$\begin{aligned}
&\texttt{delivery}(\texttt{Pno}):\texttt{Days} \leftarrow && \texttt{basic}(\texttt{Pno}, \texttt{Days}).\\
&\texttt{delivery}(\texttt{Part}):\texttt{Days} \leftarrow && \texttt{assb}(\texttt{Part}, \texttt{Sub}, _), \texttt{Days}:[\texttt{delivery}(\texttt{Sub})].\\
&\texttt{actualDays}(\texttt{Part}, \texttt{CDays}) \leftarrow && \texttt{CDays} =![\texttt{delivery}(\texttt{Part})].
\end{aligned}$$

For each assembled part, we find each basic subpart along with the number of days this takes to arrive. Observe that the argument `Pno` is projected out, and only the number of days associated with it is retained, whereby the maximum of the number of days required by any basic part is derived.

4 Advanced Graph Applications

We now show some examples that use Datalog^{FS} with positive rational numbers. As explained in more details in [11], we can assume that we use rational numbers with the same large denominator, and thus easily derive equivalent Datalog^{FS} rules for their numerators.

Diffusion Models. The Jackson-Yariv Diffusion Model (JYDM) [13] provides a powerful abstraction on how social structures influence the spread of a behavior and trends in Social Networks. We use the JDYM to understand how a tweet will spread in the Twitter network. Let $\texttt{followd}(\texttt{X}, \texttt{Y})$ indicate that user `X` is followed by user `Y` and $\texttt{coeff}(\texttt{X}, \texttt{C})$ means that `C` is the coefficient of how susceptible to change node `X` is. Predicate $\texttt{b}(\texttt{X})$ denotes, if true, that node `X` will retweet. We assume that an agent, $\texttt{source}(\texttt{X})$, first posts the tweet and starts its diffusion.

Example 11. Modeling a retweet in Datalog^{FS}.

$$\begin{aligned}
&\texttt{b}(\texttt{X}) \leftarrow \texttt{source}(\texttt{X}).\\
&\texttt{b}(\texttt{X}) \leftarrow \texttt{coeff}(\texttt{X}, \texttt{C}),\ \texttt{K} \geq \texttt{1/C},\ \texttt{K}:[\texttt{follwd}(\texttt{Y}, \texttt{X}), \texttt{b}(\texttt{Y})].
\end{aligned}$$

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

The last rule finds each node X for which the condition $\mathtt{K} \times \mathtt{C} \geq \mathtt{1}$ holds by testing the equivalent condition $\mathtt{K} \geq \mathtt{1/C}$. When this is satisfied $\mathtt{b(X)}$ is set to true. The answer to the query ?$\mathtt{b(Y)}$ will thus list all the users that propagated the tweet that originated from the node specified by `source`.

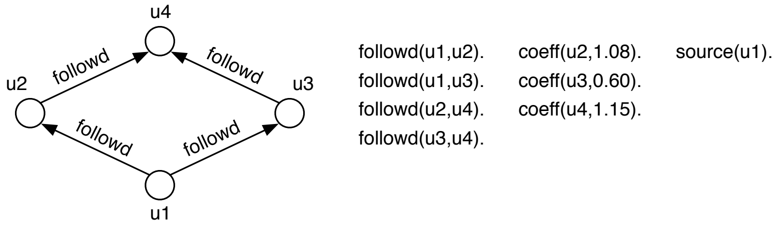

Fig. 1. Retweet modeling in Datalog^{FS}.

Then, applying the program in Example 11 to the Twitter network in Figure 1 the following atoms are derived: $\mathtt{b(u_1), b(u_2), b(u_4)}$.

Markov Chains and Page Rank. A Markov chain is represented by the transition matrix W of $s \times s$ components where w_{ij} is the probability to go from state i to state j in one step. A Markov chain is *irreducible* if for each pair of states i, j, the probabilities to go from i to j and from j to i in one or more steps is greater than zero.

Computing stabilized probabilities of a Markov chain has many real-world applications, such as estimating the distribution of population in a region, and determining the Page Rank of web nodes. Let P be a vector of stabilized probabilities of cardinality s, the equilibrium condition in terms of matrices is: $P = W \cdot P$.

Computing this fixpoint is far from trivial and irreducible chains can be modeled quite naturally in Datalog^{FS}. If $\mathtt{p\_state(X)}:\mathtt{K}$ denotes that $\mathtt{K}$ is the probability of node $\mathtt{X}, 1 \leq \mathtt{X} \leq \mathtt{s}$, and $\mathtt{w\_matrix(Y, X)}:\mathtt{W}$ denotes that the arc from $\mathtt{Y}$ to $\mathtt{X}$ has weight $\mathtt{W}$, then we compute the fixpoint as follows:

$$
\begin{array}{ll}
\mathtt{p\_state(X)\!:\!K} \leftarrow & \mathtt{K\!:\![p\_state(Y), w\_matrix(Y,X)]}. \\
\mathtt{w\_matrix(1,1)\!:\!w_{11}}. & \\
\mathtt{w\_matrix(1,2)\!:\!w_{12}}. & \\
\vdots & \\
\mathtt{w\_matrix(s,s)\!:\!w_{ss}}. &
\end{array}
$$

It is important to notice that each fixpoint of such program is an equilibrium $P = W \cdot P$ of the Markov Chain represented by matrix W. In order to find a non trivial fixpoint ($\neq 0$) for program P, we add baseline facts i.e. a set of facts $\mathtt{p\_state(1)} : 0.1.$ $\mathtt{p\_state(2)} : 0.1.$ $\ldots$ $\mathtt{p\_state(s)} : 0.1.$, that guarantee that the least fixpoint contains facts with predicate `p_state`. Such program is called *Pbl* and is a Datalog^{FS} program for which we can compute the least fixpoint efficiently. Moreover, every fixpoint of *Pbl* is also a fixpoint for P. Indeed, for any interpretation I that contains all the baseline facts, the application of either operators produce the same result: i.e., $T_P(I) = T_{Pbl}(I)$. Therefore any fixpoint of T_P that contains all the baseline facts is also a fixpoint for T_{Pbl} and vice-versa.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

But since, by its very definition, the least model of *Pbl* contains all the baseline facts, we have that every fixpoint for T_{Pbl} is also a fixpoint for T_P. The opposite of course is not true since the null fixpoint of T_P, and possibly others, are not fixpoint for T_{Pbl}. However, if T_P has a fixpoint that is positive at all nodes, then by multiplying the frequency at all nodes by a large enough finite constant, we obtain a fixpoint for T_P that contains all the baseline facts of T_{Pbl}. Since, for each irreducible Markov chain there exists a not trivial fixpoint, also T_P has one that is not null at every node, then there exists a finite fixpoint for T_{Pbl}. Therefore, the least fixpoint for T_{Pbl} is finite. That is:

Theorem 1.

- *The least fixpoint of the baseline Datalog^{FS} program that models an irreducible Markov chain is finite.*
- *Every non-null solution of an irreducible Markov chain can be obtained by scaling the least fixpoint solution of its baseline Datalog^{FS} model.*

In summary, while there has been a significant amount of previous work on Markov chains, the use of Datalog^{FS} has provides us with a model and a simple computation algorithm which is valid for all irreducible Markov chains, including periodic ones.

5 Efficient Implementation

The greater expressive power of Datalog^{FS} combines with its amenability to efficient implementation via the following three optimization steps: (i) differential fixpoint, (ii) Magic Sets, and (iii) Max-optimization. Since (ii) is basically the same as that in Datalog [11], we will discuss here (i) and (iii).

In Datalog^{FS} the differential fixpoint step is applied to recursive rules after they are transformed to ensure that every goal in the b-expression also appears outside the bracket. To satisfy this requirement, the second rule in Example 7 is transformed by repeating outside the bracket the two clauses inside the bracket, producing the following rule:

$$\mathtt{join(Y)} \leftarrow \mathtt{requires(Y,K)}, \mathtt{friend(Y,X1)}, \mathtt{join(X1)}, \mathtt{K}\!:\![\mathtt{friend(Y,X)}, \mathtt{join(X)}].$$

Since we have renamed the local variables ($\mathtt{X}$ for the case at hand), and since $\mathtt{K} \geq 1$, this transformation does not change the meaning of the rule. However it greatly simplifies its symbolic differentiation since the bracketed expression can now be treated as a constant. Thus the rule becomes linear and its δ version is:

$$\delta\mathtt{join(Y)} \leftarrow \mathtt{requires(Y,K)}, \mathtt{friend(Y,X1)}, \delta\mathtt{join(X1)}, \mathtt{K}\!:\![\mathtt{friend(Y,X)}, \mathtt{join(X)}].$$

The Max-optimization transforms the delta rules so obtained by replacing the running FS-construct with the final FS-construct. For instance, we start by rewriting the delta rule above into the following one that preserves its operational semantics:

$$\begin{aligned}\delta\mathtt{join(Y)} \leftarrow\; & \mathtt{requires(Y,K1)}, \mathtt{friend(Y,X1)}, \delta\mathtt{join(X1)},\\ & \mathtt{K}\!:\![\mathtt{friend(Y,X)}, \mathtt{join(X)}], \mathtt{K} \geq \mathtt{K1}.\end{aligned}$$

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

Now, we can replace the running-FS `K`:[...] by the final-FS `K` =![...] and still preserve the operational semantics of the rule, due to the fact that the rule only uses `K` in monotonic functions and predicates (e.g., predicates that if they are true for `K` they are also true for every value larger than `K`). This optimization would not be possible if the body uses some non-monotonic predicate, e.g., a goal that checks that `K` is even. A similar situation occurs in Examples 9, where instead of the running-FS construct in the recursive δ rules we can use the final-FS construct. Indeed the values produced by the former satisfy the external goal `C` =![`cbasic(part)`] iff the values produced by the latter do. Again this equivalence is due to the monotoniciy of the arithmetic and boolean predicates used in the rule. Such monotonicity holds in all examples given in this paper and the many examples of practical interest discussed in [11].

6 Conclusion

In this paper, we studied the important problem of allowing aggregates in recursive Datalog rules, and proposed a solution of surprising simplicity for this long-standing challenge. Our Datalog^{FS} approach is based on using continuous aggregate-like functions that allow us to query and reason on the frequency with which predicates and conjunctions of predicates occur.

References

1. J. M. Hellerstein. Datalog redux: experience and conjecture. In *PODS*, pages 1–2, 2010.
2. G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: Rewriting and optimization. In *ICDE*, pages 2–13, 2011.
3. F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *EDBT*, pages 1–8, 2011.
4. C. Zaniolo. The logic of query languages for data streams. In *Logic and Databases 2011. EDBT 2011 Workshops*, pages 1–2, 2011.
5. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
6. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.
7. I. S. Mumick and O. Shmueli. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence*, 15:407–435, 1995.
8. S. Greco and C. Zaniolo. Greedy algorithms in datalog. *TPLP*, 1(4):381–407, 2001.
9. F. Arni, K. Ong, S. Tsur, H. Wang, and C. Zaniolo. The deductive database system ldl++. *TPLP*, 3(1):61–94, 2003.
10. I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB*, pages 264–277, 1990.
11. M. Mazuran, E. Serra, and C Zaniolo. Graph languages in Datalog^{FS}: from abstract semantics to efficient implementation. Technical report, UCLA, 2011.
12. J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition.* Springer, 1987.
13. M. O. Jackson and L. Yariv. Diffusion on social networks. *Economie Publique*, 2005.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,