Nondeterministic, Nonmonotonic Logic Databases

Fosca Giannotti, Giuseppe Manco, Mirco Nanni, and Dino Pedreschi

Abstract—We consider in this paper an extension of Datalog with mechanisms for temporal, nonmonotonic, and nondeterministic reasoning, which we refer to as Datalog++. We show, by means of examples, its flexibility in expressing queries concerning aggregates and data cube. Also, we show how iterated fixpoint and stable model semantics can be combined to the purpose of clarifying the semantics of Datalog++ programs and supporting their efficient execution. Finally, we provide a more concrete implementation strategy on which basis the design of optimization techniques tailored for Datalog++ is addressed.

Index Terms—Logic programming, databases, negation, nondeterminism, stable models.

1 Introduction

1.1 Motivations

The name **Datalog++** is used in this paper to refer to Datalog extended with mechanisms supporting:

- a limited form of *temporal* reasoning by means of temporal, or *stage*, arguments of relations, ranging over a discrete temporal domain, in the style of [5];
- *nonmonotonic* reasoning by means of a form of stratified negation w.r.t. the stage arguments, called *XY-stratification* [26];
- *nondeterministic* reasoning by means of the nondeterministic `choice` construct [12].

Datalog++, which is essentially a fragment of $\mathcal{LDL}++$ [2], and is advocated in [27, chapter 10], revealed a highly expressive language, with applications in diverse areas such as AI planning [4], active databases [25], object databases [8], semistructured information management and Web restructuring [10], data mining and knowledge discovery in databases [15], [3], [11]. However, a thorough study of the semantics of Datalog++ is still missing, which provides a basis for sound and efficient implementations and optimization techniques. A preliminary study of the semantics of Datalog++ is sketched in [4] by discussing the relation between a declarative (model-theoretic) semantics and a fixpoint (bottom-up) semantics. However, this discussion is informal and, moreover, fails to achieve a precise coincidence of the two semantics for arbitrary programs. Therefore, we found motivation in an in-depth study of the semantics of Datalog++ programs, which also explains the reason for the missing coincidence of semantics in [4] and proposes a general condition which ensures such coincidence.

- *F. Giannotti and G. Manco are with CNUCE Institute of the CNR, Via S. Maria 36, 56125 Pisa, Italy. E-mail: {F.Giannotti, G.Manco}@cnuce.cnr.it.*
- *M. Nanni and D. Pedreschi are with the Dipartimento di Informatica, Università di Pisa, C.so Italia 40, 56125 Pisa, Italy. E-mail: {nanni, pedre}@di.unipi.it.*

Manuscript received 29 July 1998; revised 4 Nov. 1999; accepted 13 Dec. 1999; posted to Digital Library 6 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 107188.

1.2 Objectives and Contributions

This paper is aimed at

1. Illustrating the expressiveness and flexibility of Datalog++ as a query language,
2. Providing a declarative semantics for Datalog++ which integrates the temporal, nonmonotonic, and nondeterministic mechanisms and which justifies the adoption of an iterated fixpoint semantics for the language, and
3. Providing the basis for query optimization in Datalog++, thus making it viable as an efficient implementation.

To this purpose, we proceed as follows:

1. The use of Datalog++ as a query language is discussed in Section 2.
2. A natural, purely declarative, semantics for Datalog++ is assigned using the notion of a *stable model* in Section 3; a constructive semantics is then assigned using an iterative procedure which exploits the stratification induced by the progression of the temporal argument.
3. In the main result of this paper, we show that the two semantics are equivalent, provided that a natural syntactic restriction is fulfilled, which imposes a disciplined use of the temporal argument within the *choice* construct.
4. On the basis of this result, we introduce, in Section 4, a more concrete operational semantics using relational algebra operators and, in Section 5, a repertoire of optimization techniques, especially tailored for Datalog++. In particular, we discuss how it is possible to support efficient history-insensitive temporal reasoning by means of real side effects during the iterated computation [19].

The material in Sections 2 and 3 is based on results presented in a more compact form in [9], [14], while the material in Sections 4 and 5 is new. In conclusion, this paper provides a thorough account of the pragmatics, semantics, and implementation of Datalog++.

1.3 Related Work

Nondeterminism is introduced in deductive databases by means of the `choice` construct. The original proposal in [17] was later revised in [23] and refined in [12]. These studies exposed the close relationship connecting nonmonotonic reasoning with nondeterministic constructs, leading to the definition of a stable model semantics for `choice`. While the declarative semantics of `choice` is based on *stable model* semantics, which is intractable in general, `choice` is amenable to efficient implementations and it is actually supported in the logic database language $\mathcal{LDL}$ [20] and its evolution $\mathcal{LDL}++$ [2].

On the other side, stratification has been a crucial notion for the introduction of nonmonotonic reasoning in deductive databases. From the original idea in [1] of a static stratification based on predicate dependencies, stratified negation has been refined to deal with dynamic notions, as in the case of locally stratified programs [21] and modularly stratified programs [22]. Dynamic, or local, stratification has a close connection with temporal reasoning, as the progression of time points yields an obvious stratification of programs—consider, for instance, $\mathrm{Datalog}_{1S}$ [5]. It is therefore natural that nonmonotonic and temporal reasoning are combined in several deductive database languages, such as those in [18], [16], [10], [27, chapter 10].

However, a striking mismatch is apparent between the above two lines of research: Nondeterminism leads to a multiplicity of (stable) models, whereas stratification leads to a unique (perfect) model. So far, no comprehensive study has addressed the combination of the two lines which occurs in Datalog++ and which requires the development of a *nondeterministic iterated fixpoint* procedure. We notice, however, the mentioned exception of [4], where an approach to this problem is sketched with reference to locally stratified programs augmented with `choice`. In the present paper, we present instead a thorough treatment of Datalog++ programs and repair an inconvenience of the approach in [4] concerning the incompleteness of the iterated fixpoint procedure.

1.4 Preliminaries

We assume that the reader is familiar with the concepts of relational databases and of the Datalog language [24]. Various extensions of Datalog are considered in this paper:

- *Datalog*$\neg$ is the language with unrestricted use of negation in the body of the rules.
- *Datalog*$\neg_s$ is the subset of *Datalog*$\neg$ consisting of *(predicate-)stratified programs*, in the sense of [1].
- *Datalog*$_{1S}$ is the language introduced in [5], where each predicate may have a designated argument, called a *stage argument*, ranging over natural numbers (where each natural number n is represented by the term $s^n(nil)$).
- *Datalog*$^{\neg}_{1S}$ is the language *Datalog*$_{1S}$ with an unrestricted use of negation in rule bodies.
- *Datalog++* is the subset of *Datalog*$^{\neg}_{1S}$ where negation is used in a disciplined manner, according to the mechanisms of *XY-stratification* and *nondeterministic choice*, which are introduced in the next section.

2 Query Answering with Datalog++

Datalog, the basis of deductive databases, is essentially a friendly syntax expressing relational queries and extending the query facilities of the relational calculus with recursion. Datalog's simplicity in expressing complex queries impacted on the database technology and, nowadays, recursive queries/views has become part of the SQL3 standard. Recursive queries find natural applications in all areas of information systems where computing transitive closures or traversals is an issue, such as in bill-of-materials queries, route, or plan formation, graph traversals, etc.

However, it is widely recognized that the expressiveness of Datalog's (recursive) rules is limited and several extensions, along various directions, have been proposed. In this paper, we address, in particular, two such directions, namely, nondeterministic and nonmonotonic reasoning, supported, respectively, by the `choice` construct and the notion of XY-stratification. We introduce these mechanisms by means of a few examples, which are meant to point out the enhanced query capabilities.

2.1 Nondeterministic Choice

The `choice` construct is used to nondeterministically select subsets of answers to queries which obey a specified FD constraint. For instance, the rule

$$\texttt{st\_ad}(\texttt{St}, \texttt{Ad}) \leftarrow \texttt{major}(\texttt{St}, \texttt{Area}),\ \texttt{faculty}(\texttt{Ad}, \texttt{Area}), \texttt{choice}((\texttt{St}), (\texttt{Ad}))$$

assigns to each student a unique, arbitrary advisor from the same area since the `choice` goal constrains the `st_ad` relation to obey the FD (`St` $\rightarrow$ `Ad`). Therefore, if the base relation `major` is formed by the tuples {<smith, db>, <gray, se>} and the base relation `faculty` is formed by the tuples {<brown, db>, <scott, db>, <miller, se>}, then there are two possible outcomes for the query `st_ad(St,Ad)`: either {<smith, brown>, <gray, miller>} or {<smith, scott>, <gray, miller>}. In practical systems, such as $\mathcal{LDL}++$, one of these two solutions is computed and presented as a result.

Thus, a first use of `choice` is in computing nondeterministic, nonrecursive queries. However, `choice` can be combined with recursion, as in the following rules which compute an arbitrary ordering of a given relation `r`:

$$\texttt{ord\_r}(\texttt{root}, \texttt{root}).$$
$$\texttt{ord\_r}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{ord\_r}(\_, \texttt{X}), \texttt{r}(\texttt{Y}), \texttt{choice}((\texttt{X}), (\texttt{Y})), \texttt{choice}((\texttt{Y}), (\texttt{X})).$$

Here, `root` is a fresh constant, conveniently used to simplify the program. If the base relation `r` is formed by k tuples, then there are $k!$ possible outcomes for the query $\texttt{ord\_r}(\texttt{X}, \texttt{Y})$, namely, a set:

$$\{\texttt{ord\_r}(\texttt{root}, \texttt{root}), \texttt{ord\_r}(\texttt{root}, \texttt{t}_1), \texttt{ord\_r}(\texttt{t}_1, \texttt{t}_2), \ldots, \texttt{ord\_r}(\texttt{t}_{k-1}, \texttt{t}_k)\}$$

for each permutation $\{\texttt{t}_1, \ldots, \texttt{t}_k\}$ of the tuples of `r`. Therefore, in each possible outcome of the mentioned query, the relation `ord_r` is a total ordering of the tuples of `r`. The double `choice` constraint in the recursive rule specifies that the successor and predecessor of each tuple of `r` is unique.

Interestingly, choice can be employed to compute new *deterministic* queries, which are inexpressible in Datalog, as well as in pure relational calculus. A remarkable example is the capability of expressing *aggregates*, as in the following program which computes the summation aggregate over a relation r, which uses an arbitrary ordering of r computed by ord_r:

$$\texttt{sum\_r}(\texttt{root}, 0).$$
$$\texttt{sum\_r}(\texttt{Y}, \texttt{N}) \leftarrow \texttt{sum\_r}(\texttt{X}, \texttt{M}), \texttt{ord\_r}(\texttt{X}, \texttt{Y}), \texttt{N} = \texttt{M} + \texttt{Y}.$$

$$\texttt{total\_sum\_r}(\texttt{N}) \leftarrow \texttt{sum\_r}(\texttt{X}, \texttt{N}), \neg \texttt{ord\_r}(\texttt{X}, \_).$$

Here, sum_r(X,N) is used to accumulate in N the summation up to X, with respect to the order given by ord_r. Therefore, the total sum is reconstructed from $\texttt{sum\_r}(\texttt{X}, \texttt{N})$ when X is the last tuple in the order. Notice the use of (stratified) negation to the purpose of selecting the last tuple. In practical languages, such as $\mathcal{LDL}++$, some syntactic sugar for aggregation is used as an abbreviation of the above program [26]:

$$\texttt{total\_sum\_r}(\texttt{sum} < \texttt{X} >) \leftarrow \texttt{r}(\texttt{X}).$$

On the basis of this simple example, more sophisticated forms of aggregation, such as *datacube* and other OLAP functions, can be built. As an example, consider a relation sales(Date, Department, Sale) and the problem of aggregating sales along the dimensions Date and Department. Three aggregation patterns are then possible, corresponding to the various facets of the datacube: <Date,*>, <*,Department>, <*,*>. The former two patterns correspond to the aggregation of sales along a single dimension (respectively, Department and Date), and can be obtained from the original relation by applying the method shown above. The latter pattern, then, can be obtained by recursively applying such a method to one of the two patterns previously computed in order to aggregate along the remaining dimension. In case there are several dimensions along which to aggregate, we can simply repeat the process, aggregating at each step along a new (i.e., still nonaggregated) dimension.

A thorough account on programming with nondeterminism in deductive databases can be found in [7], [13].

The semantics of choice is assigned using the so-called *stable model* semantics of Datalog¬ programs, a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages by Gelfond and Lifschitz [6]. To define the notion of a stable model, we need to introduce a transformation H which, given an interpretation I, maps a Datalog¬ program P into a positive Datalog program $H(P, I)$:

$$H(P, I) = \{A \leftarrow B_1, \ldots, B_n \mid A \leftarrow B_1, \ldots, B_n, \\ \neg C_1, \ldots, \neg C_m \in \mathit{ground}(P) \wedge \{C_1, \ldots, C_m\} \cap I = \emptyset\}.$$

Next, we define:

$$S_P(I) = T_{H(P,I)} \uparrow \omega.$$

Then, M is said to be a *stable model* of P if $S_P(M) = M$. In general, Datalog¬ programs may have zero, one, or many stable models. The multiplicity of stable models can be exploited to give a declarative account of nondeterminism.

We can in fact define the *stable version* of a program:[1]

Definition 1 (Stable Version). *Given a Datalog¬ program P, its stable version, $SV(P)$, is defined as the program obtained from P by replacing all the references to the* choice *atom in a rule $r : H \leftarrow B, \texttt{choice}((\texttt{X}), (\texttt{Y}))$ (X and Y are disjunct vectors of variables that also appear in B) with the atom $\texttt{chosen}_\texttt{r}(\texttt{X}, \texttt{Y})$. The $\texttt{chosen}_\texttt{r}$ predicate is defined by the following rules:*

$$\texttt{chosen}_\texttt{r}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{B}, \neg \texttt{diffchoice}_\texttt{r}(\texttt{X}, \texttt{Y}).$$
$$\texttt{diffchoice}_\texttt{r}(\texttt{X}, \texttt{Y}) \leftarrow \texttt{chosen}_\texttt{r}(\texttt{X}, \texttt{Y}'), \texttt{Y} \neq \texttt{Y}'.$$

In the above definition, for any fixed value of X, each choice for Y inhibits all the other possible ones via $\texttt{diffchoice}_\texttt{r}$ so that, in the stable models of $SV(P)$, there is (only) one of them. Notice that, by construction, each occurrence of a choice atom has its own pair of chosen and diffchoice atoms, thus bounding the scope of the atom to the rule it appears in. The various stable models of the transformed program $SV(P)$ thus correspond to the choice models of the original program.

2.1.1 XY-Programs

Another notion used in this paper is that of XY-programs originally introduced in [26]. The language of such programs is $Datalog^{\neg}_{1S}$, which admits negation on body atoms and a unary constructor symbol, used to represent a temporal argument usually called the *stage argument*. A general definition of XY-programs is the following:

Definition 2 (XY-Stratification). *A set P of $Datalog^{\neg}_{1S}$ rules defining mutually recursive predicates is an XY-program if it satisfies the following conditions:*

1. *Each recursive predicate has a distinguished stage argument.*
2. *Every recursive rule r is either an X-rule or a Y-rule, where:*
   - *r is an X-rule when the stage argument in every recursive predicates in r is the same variable,*
   - *r is a Y-rule when 1) the head of r has a stage argument $s(J)$, where J is a variable, 2) some goal of r has J as its stage argument, and 3) the remaining recursive goals have either J or $s(J)$ as their stage argument.*

Intuitively, in the rules of XY-programs, an atom $\texttt{p}(\texttt{J}, \_)$ denotes the extension of relation p at the current stage (present time) J, whereas an atom $\texttt{p}(\texttt{s}(\texttt{J}), \_)$ denotes the extension of relation p at the next stage (future time) s(J). By using a different *primed* predicate symbol p′ in the $\texttt{p}(\texttt{s}(\texttt{J}), \_)$ atoms, we obtain the so-called *primed version* of an XY-program. We say that an XY-program is *XY-stratified* if its primed version is a stratified program. Intuitively, if the dependency graph of the primed version has no cycles through negated edges, then it is possible to obtain an

1. For ease of presentation, here we include the definition of $SV(P)$ for the case of at most one choice atom per rule. The definition for the general case can be found in [13].

ordering on the original rules modulo the stage arguments. As a consequence, an XY-stratified program is also locally stratified and therefore has a unique stable model that coincides with its perfect model [21].

Let P be an XY-stratified program. Then, for each $i > 0$, define P_i as

$$P_i = \{r[s^i(nil)/I] \mid r \in P, \\ I \text{ is the stage argument of the head of } r\}$$

(here, $r[x/I]$ stands for r where I is replaced by x), i.e., P_i is the set of rule instances of P that define the predicates with stage argument $s^i(nil) = i$. Then, the iterated fixpoint procedure for computing the (unique) minimal model of P can be defined as follows:

1. Compute M_0 as the minimal model of P_0.
2. For each $j > 0$, compute M_j as the minimal model of $P_j \cup M_{j-1}$.

Notice that, for each $j \geq 0$, P_j is stratified by the definition and, hence, its perfect model M_j is computable via an iterated fixpoint procedure.

In this paper, we use the name Datalog++ to refer to the language of XY-programs augmented with `choice` goals.

3 A Semantics for Datalog++

When `choice` constructs are allowed in XY-programs, a multiplicity of stable models exists for any given program and, therefore, it is needed to clarify how this phenomenon combines with the iterated fixpoint semantics of `choice`-free XY-programs. This task is accomplished in three steps.

1. First, we present a general result stating that whenever a *Datalog¬* program P is stratifiable into a hierarchy of recursive cliques (i.e., minimal sets of mutually recursive rules) $Q_1, Q_2, \ldots$, then any stable model of the entire program P can be reconstructed by iterating the construction of approximating stable models, each associated to a clique.
2. Second, we observe that, under a syntactic restriction on the use of the `choice` construct that does not compromise expressiveness, Datalog++ programs can be naturally stratified into a hierarchy of recursive cliques $Q_1, Q_2, \ldots$ by using the temporal arguments of recursive predicates.
3. Third, by the observation in 2, we can apply the general result in 1 to Datalog++ programs, thus obtaining that the stable models of the entire program can be computed by an iterative fixpoint procedure which follows the stratification induced by the temporal arguments.

Given a (possibly infinite) program P, consider a (possibly infinite) topological sort of its distinct recursive cliques $Q_1 \prec Q_2 \ldots \prec Q_i \prec \ldots$ induced by the dependency relation over the predicates of P. Given an interpretation I, we use the notation I_i to denote the subset of atoms of I whose predicate symbols are predicates defined in clique Q_i.

The following observations are straightforward:

- $\bigcup_{i>0} I_i = I$ and, analogously, $\bigcup_{i>0} Q_i = P$.
- The predicates defined in Q_{i+1} depend only on the definitions in $Q_1 \cup \ldots \cup Q_i$; as a consequence, the interpretation of Q_{i+1} is $I_1 \cup \ldots \cup I_i \cup I_{i+1}$ (i.e., we can ignore $\bigcup_{j>i+1} I_j$).

The next definition shows how to transform each recursive clique, within the given topological ordering, in a self-contained program which takes into account the information deduced by the previous cliques. Such transformation resembles the Gelfond-Lifschitz transformation reported in Section 2.

Definition 3. *Consider a program P, a topological sort of its recursive cliques $Q_1 \prec Q_2 \ldots \prec Q_i \ldots$ and an interpretation $I = \bigcup_{i>0} I_i$. Now, define*

$$Q_i^{red(I)} = \{H \leftarrow B_1, \ldots, B_n \mid H \leftarrow B_1, \ldots, B_n, \\ C_1, \ldots, C_m \in ground(Q_i) \\ \wedge\, B_1, \ldots, B_n \text{ are defined in } Q_i \\ \wedge\, C_1, \ldots, C_m \text{ are defined in } (Q_1 \cup \ldots \cup Q_{i-1}) \\ \wedge\, I_1 \cup \ldots \cup I_{i-1} \models C_1, \ldots, C_m\}.$$

The idea underlying the transformation is to remove from each clique Q_i all the dependencies induced by the predicates which are defined in lower cliques. We abbreviate $Q_i^{red(I)}$ by Q_i^{red}, when the interpretation I is clear by the context.

Example 1. Consider the program $P = \{\mathtt{p} \leftarrow \mathtt{q}, \mathtt{r}.\ \mathtt{q} \leftarrow \mathtt{r}, \mathtt{t}.\ \mathtt{r} \leftarrow \mathtt{q}, \mathtt{s}.\}$ and the recursive cliques $Q_1 = \{\mathtt{q} \leftarrow \mathtt{r}, \mathtt{t}.\ \mathtt{r} \leftarrow \mathtt{q}, \mathtt{s}.\}$ and $Q_2 = \{\mathtt{p} \leftarrow \mathtt{q}, \mathtt{r}.\}$. Now, consider the interpretation $I = \{\mathtt{s}, \mathtt{q}, \mathtt{r}\}$. Then, $Q_1^{red} = \{\mathtt{q} \leftarrow \mathtt{r}, \mathtt{t}.\ \mathtt{r} \leftarrow \mathtt{q}, \mathtt{s}.\}$ and $Q_2^{red} = \{\mathtt{p} \leftarrow .\}$.

The following Lemma 1 states the relation between the models of the transformed cliques and the models of the program. We abbreviate $I_1 \cup \ldots \cup I_i$ with $I^{(i)}$ and analogously for $Q^{(i)}$.

Lemma 1. *Given a (possibly infinite) Datalog¬ program P and an interpretation I, let $Q_1 \prec Q_2 \ldots \prec Q_i \ldots$ and $I_1 \prec I_2 \ldots \prec I_i \ldots$ be the topological sorts on P and I induced by the dependency relation of P. Then, the following statements are equivalent:*

1. $S_P(I) = I$
2. $\forall i > 0. S_{Q_i^{red}}(I_i) = I_i$
3. $\forall i > 0. S_{Q^{(i)}}(I^{(i)}) = I^{(i)}$

Proof Sketch. The proof is structured as follows: $1 \Longleftrightarrow 3$ and $2 \Longleftrightarrow 3$.

- $3 \Longrightarrow 1$ We next show that a) $S_P(I) \subseteq I$ and b) $I \subseteq S_P(I)$.
  a. Each rule in $H(P, I)$ comes from a rule r of P, which in turn appears in $Q^{(i)}$ for some i and, then, $I^{(i)}$ is a model of r, by the hypothesis. No atom in $I \setminus I^{(i)}$ appears in r, so, also, I is model of r. I is then a model of $H(P, I)$ and, hence, $S_P(I) \subseteq I$.
  b. If $A \in I$, then $A \in I^{(i)}$ for some i, so (by the hypothesis and definition of S_P) for each I^* such that $I^* = T_{H(Q^{(i)}, I^{(i)})}(I^*)$, $A \in I^*$. Moreover, for each I' such that $I' = T_{H(P,I)}(I')$, it is

readily checked that, for each i, $I'^{(i)} = T_{H(Q^{(i)},I^{(i)})}(I'^{(i)})$ and then $I \subseteq S_P(I)$.

- $1 \Longrightarrow 3$ We observe that $I = min\{I^* \mid I^* = T_{H(P,I)}(I^*)\}$, which implies: $I^{(i)} = min\{I^{*^{(i)}} \mid I^{*^{(i)}} = T_{H(Q^{(i)},I^{(i)})}(I^{*^{(i)}})\}$.
- $2 \Longrightarrow 3$ We proceed by induction on i. The base case is trivial. In the inductive case, we next show that a) $S_{Q^{(i)}}(I^{(i)}) \subseteq I^{(i)}$ and b) vice versa.

  a. Notice that, from the induction hypothesis, $I^{(i)} \models Q^{(i-1)}$ and then it suffices to show ithat $I^{(i)} \models Q_i$ (by a simple case analysis).
  b. Exploiting the induction hypothesis, we have

$$I^{(i-1)} \subseteq S_{Q^{(i-1)}}(I^{(i-1)}) = S_{Q^{(i-1)}}(I^{(i)}) \subseteq S_{Q^{(i)}}(I^{(i)})$$

  (by definition of $H(P,I)$). We now show by induction on n that $\forall n \geq 0\ T^n_{H(Q_i^{red},I_i)} \subseteq T^\omega_{H(Q^{(i)},I^{(i)})}$. The base case $n = 0$ is trivial. In the induction case ($n > 0$), if $A \in T^n_{H(Q_i^{red},I_i)}$, then there exists a rule $A \leftarrow b_1, \ldots, b_h$ in $H(Q_i^{red}, I_i)$ such that $\{b_1, \ldots, b_h\} \subseteq T^{n-1}_{H(Q_i^{red},I_i)}$. Now, by definition of H and Q_i^{red}, there exists a rule:

$$A \leftarrow b_1, \ldots, b_h, \neg c_1, \ldots, \neg c_j, d_1, \ldots, d_k, \neg e_1, \ldots, \neg e_l$$

  in Q_i such that $\{c_1, \ldots, c_j\} \cap I_i = \emptyset$ and $I^{(i-1)} \models d_1 \wedge \ldots \wedge d_k \wedge \neg e_1 \wedge \ldots \wedge \neg e_l$. Observe now that, by definition of

$$H, A \leftarrow b_1, \ldots, b_h, d_1, \ldots, d_k \in H(Q^{(i)}, I^{(i)}).$$

  Furthermore, by the induction hypothesis and $I^{(i-1)} \subseteq S_{Q^{(i)}}(I^{(i)})$, we have the following: $\{b_1, \ldots, b_h, d_1, \ldots, d_k\} \subseteq T^\omega_{H(Q^{(i)},I^{(i)})}$. Hence, by definition of T^ω, $A \in T^\omega_{H(Q^{(i)},I^{(i)})}$, that is, $A \in S_{Q^{(i)}}(I^{(i)})$. This completes the innermost induction and we obtain that $I_i = S_{Q_i^{red}}(I_i) \subseteq S_{Q^{(i)}}(I^{(i)})$.
- $3 \Longrightarrow 2$ We proceed is a way similar to the preceding case. To see that $\forall i\ I_i \subseteq S_{Q_i^{red}}(I_i)$, it suffices to verify that, for each rule instance r with head A, the following property holds: $\forall n\ A \in T^n_{H(Q^{(i)},I^{(i)})} \Rightarrow A \in T^n_{H(Q_i^{red},I_i)}$. For the converse, we simply observe that I_i is a model of Q_i^{red}. □

This result states that an arbitrary Datalog¬ program has a stable model if and only if each of its approximating cliques, according to the given topological sort, has a *local* stable model. This result gives us an intuitive idea for computing the stable models of an approximable program by means of the computation of the stable models of its approximating cliques.

Notice that Lemma 1 holds for arbitrary programs provided that a stratification into a hierarchy of cliques is given. In this sense, this result is more widely applicable than the various notions of stratified programs, such as that of *modularly stratified programs* [22], in which it is required that each clique Q_i^{red} is locally stratified. On the contrary, here we do not require that each clique is, in any sense, stratified. This is motivated by the objective of dealing with nondeterminism and justifies why we adopt the (nondeterministic) stable model semantics, rather than other deterministic semantics for (stratified) Datalog¬ programs, such as, for instance, perfect model semantics [21].

We turn now our attention to XY-programs. The result of instantiating the clauses of an XY-program P with all possible values (natural numbers) of the stage argument yields a new program, $SG(P)$ (for *stage ground*). More precisely, $SG(P) = \bigcup_{i\geq 0} P_i$, where

$$P_i = \{r[i/I] \mid r \text{ is a rule of } P, I \text{ is the stage argument of } r\}.$$

The stable models of P and $SG(P)$ are closely related.

Lemma 2. *Let P be an XY-program. Then, for each interpretation I:*

$$S_P(I) = I \Longleftrightarrow S_{SG(P)}(I) = I.$$

Proof Sketch. We show by induction that

$$\forall n. T^n_{H(SG(P),I)}(\emptyset) = T^n_{H(P,I)}(\emptyset),$$

which implies the thesis. The base case is trivial. For the inductive case, observe that, since P is XY-stratified, if $A \in T^{n+1}_{H(P,I)}(\emptyset)$ then, for each $A \leftarrow B_1, \ldots, B_n \in H(P,I)$ such that $\{B_1, \ldots, B_n\} \in T^n_{H(P,I)}(\emptyset) = T^n_{H(SG(P),I)}(\emptyset)$, we have that $A \leftarrow B_1, \ldots, B_n \in H(SG(P), I)$.

Vice versa, if $A \in T^{n+1}_{H(SG(P),I)}(\emptyset)$, then, for each $A \leftarrow B_1, \ldots, B_n \in H(SG(P), I)$ such that

$$\{B_1, \ldots, B_n\} \in T^n_{H(SG(P),I)}(\emptyset) = T^n_{H(P,I)}(\emptyset),$$

we have $A \leftarrow B_1, \ldots, B_n \in H(P, I)$. □

However, the dependency graph of $SG(P)$ (which is obviously the same as P) does not necessarily induce a topological sort because, in general, XY-programs are not predicate-stratified and, therefore, Lemma 1 is not directly applicable. To tackle this problem, we distinguish the predicate symbol p in the program fragment P_i from the same predicate symbol in all other fragments P_j with $j \neq i$ by differentiating the predicate symbols using the temporal argument. Therefore, if $\mathtt{p(i,x)}$ is an atom involved in some rule of P_i, its modified version is $\mathtt{p_i(x)}$. More precisely, we introduce, for any XY-program P, its modified version $SO(P)$ (for *stage-out*), defined by $SO(P) = \bigcup_{i\geq 0} SO(P)_i$, where $SO(P)_i$ is obtained from the program fragment P_i of $SG(P)$ by extracting the stage arguments from any atom and adding it to the predicate symbol of the atom. Similarly, the modified version $SO(I)$ of an interpretation I is defined. Therefore, the atom $\mathtt{p(i,x)}$ is in I iff the atom $\mathtt{p_i(x)}$ is in $SO(I)$, where $\mathtt{i}$ is the value in the stage argument position of relation $\mathtt{p}$.

Unsurprisingly, the stable models of $SG(P)$ and $SO(P)$ are closely related.

Lemma 3. *Let P be an XY-program. Then, for each interpretation I:*

$$S_{SG(P)}(I) = I \Longleftrightarrow S_{SO(P)}(SO(I)) = SO(I).$$

Proof Sketch. It is easy to see that $SO(SG(P)) = SO(P)$. Hence, the least Herbrand models of $SO(H(SG(P), I))$ and $H(SO(P), SO(I))$ coincide. □

Our aim is now to conclude that, for a given Datalog++ program P:

1. $SO(P)_0 \prec SO(P)_1 \prec \cdots$ is the topological sort over $SO(P)$ in the hypothesis of Lemma 1;[2] recall that, for $i \geq 0$, the clique $SO(P)_i$ consists of the rules from $SO(P)$ with stage argument i in their heads.
2. By Lemmas 1, 2, and 3, an interpretation I is a stable model of P iff I can be constructed as $\bigcup_{i\geq 0} I_i$, where, for $i \geq 0$, I_i is a stable model of $SO(P)_i^{red(I^{(i)})}$, i.e., the clique $SO(P)_i$ reduced by substituting the atoms deduced at stages earlier than i.

On the basis of 2 above, it is possible to define an iterative procedure to construct an arbitrary stable model M of P as the union of the interpretations $M_0, M_1, \ldots$ defined as follows:

Iterated stable model procedure.

Base case. M_0 is a stable model of the bottom clique $SO(P)_0$.
Induction case. For $i > 0$, M_i is a stable model of $SO(P)_i^{red(M^{(i)})}$, i.e., the clique $SO(P)_i$ reduced with respect to $M_0 \cup \cdots \cup M_{i-1}$.

The interpretation $M = \bigcup_{i\geq 0} M_i$ is called an *iterated stable model* of P.

It should be observed that this construction is close to the procedure called *iterated choice fixpoint* in [4]. Also, following the approach of [13], each local stable model M_i can, in turn, be efficiently constructed by a nondeterministic fixpoint computation in polynomial time.

Unfortunately, the desired result that the notions of stable model and iterated stable model coincide does not hold in full generality in the sense that the iterative procedure is not complete for arbitrary Datalog++ programs, thus giving rise to the inconveniences in [4] mentioned in the introduction. In fact, as demonstrated by the example below, an undisciplined use of `choice` in Datalog++ programs may cause the presence of stable models that cannot be computed incrementally over the hierarchy of cliques.

Example 2. Consider ithe following simple Datalog++ program P:

$$\begin{aligned}&\mathtt{q(0,a)}.\\ &\mathtt{q(s(I),b)} \leftarrow \mathtt{q(I,a)}.\\ &\mathtt{p(I,X)} \leftarrow \mathtt{q(I,X)}, \mathtt{choice((),(X))}.\end{aligned}$$

In the stable version $SV(P)$ of P, the rule defining predicate p is replaced by:

$$\begin{aligned}&\mathtt{p(I,X)} \leftarrow \mathtt{q(I,X)}, \mathtt{chosen(X)}.\\ &\mathtt{chosen(X)} \leftarrow \mathtt{q(I,X)}, \neg\mathtt{diffchoice(X)}.\\ &\mathtt{diffchoice(X)} \leftarrow \mathtt{chosen(Y)}, \mathtt{Y} \neq \mathtt{X}.\end{aligned}$$

It is readily checked that $SV(P)$ admits two stable models, namely, {q(0,a), q(s(0),b), p(0,a)}, and {q(0,a), q(s(0),b), p(s(0),b)}, but only the first model is an iterated stable models and, therefore, the second model cannot be computed using the *iterated choice fixpoint* of [4].

The technical reason for this problem is that the free use of the `choice` construct inhibits the possibility of defining a topological sort on $SO(P)$ based on the value of the stage argument. In Example 2, the predicate dependency irelation of $SO(SV(P))$ induces a dependency among stage i and the stages $j > i$ because of the dependency of the `chosen` predicate from the predicates $\mathtt{q_i}$ for all stages $\mathtt{i} \geq 0$.

To prevent this problem, it suffices to require that `choice` goals refer to the stage argument I in the domain of the associated functional dependency. The Datalog++ programs which comply with this constraint are called *choice-safe*. The following is a way to turn the program of Example 2 into a choice-safe program (with a different semantics):

$$\mathtt{p(I,X)} \leftarrow \mathtt{q(I,X)}, \mathtt{choice((I),(X))}.$$

This syntactic restriction, moreover, does not greatly compromise the expressiveness of the query language in that it is possible to simulate, within this restriction, most of the general use of `choice` (see [19]).

The above considerations are summarized in the following main result of the paper, which, under the mentioned restriction of choice-safety, is a direct consequence of Lemmas 1, 2, and 3.

Theorem 1 (Correctness and Completeness of the Iterated Stable Model Procedure). *Let P be a choice-safe Datalog++ program and I an interpretation. Then, I is a stable model of $SV(P)$ iff it is an iterated stable model of P.*

The following example shows a computation with the iterated stable model procedure.

Example 3. Consider the following Datalog++ version of the *seminaive* program, discussed in [26], which nondeterministically computes a maximal path from node a over a graph g:

$$\begin{aligned}&\mathtt{delta(0,a)}.\\ &\mathtt{delta(s(I),Y)} \leftarrow \mathtt{delta(I,X)}, \mathtt{g(X,Y)},\\ &\quad \neg\mathtt{all(I,Y)}, \mathtt{choice((I,X),(Y))}.\\ &\mathtt{all(I,X)} \leftarrow \mathtt{delta(I,X)}.\\ &\mathtt{all(s(I),X)} \leftarrow \mathtt{all(I,X)}, \mathtt{delta(s(I),\_)}.\end{aligned}$$

Assume that the graph is given by $\mathtt{g} = \{\langle \mathtt{a},\mathtt{b}\rangle, \langle \mathtt{b},\mathtt{c}\rangle, \langle \mathtt{b},\mathtt{d}\rangle, \langle \mathtt{d},\mathtt{e}\rangle\}$. The following interpretations are carried out at each stage of the iterated stable model procedure:

1. $\mathtt{I}_0 = \{\mathtt{delta}_0(\mathtt{a}), \mathtt{all}_0(\mathtt{a})\}$.
2. $\mathtt{I}_1 = \{\mathtt{all}_1(\mathtt{a}), \mathtt{all}_1(\mathtt{b}), \mathtt{delta}_1(\mathtt{b})\}$.
3. $\mathtt{I}_2^1 = \{\mathtt{all}_2(\mathtt{a}), \mathtt{all}_2(\mathtt{b}), \mathtt{delta}_2(\mathtt{c}), \mathtt{all}_2(\mathtt{c})\}$, $\mathtt{I}_2^2 = \{\mathtt{all}_2(\mathtt{a}), \mathtt{all}_2(\mathtt{b}), \mathtt{delta}_2(\mathtt{d}), \mathtt{all}_2(\mathtt{d})\}$

2. In general, $SO(P)_i$ can be composed by more than one clique so that, in the above expression, it should be replaced by $SO(P)_i^1 \prec \cdots \prec SO(P)_i^{n_i}$. However, for ease of presentation, we ignore it since such a general case is trivially deducible from what follows.

$\forall p$ *defined in* Q : $P := \emptyset$;
repeat
  $\forall p$ *defined in* Q : $last\_P := P$;
    $P := EVAL(p, Rels)$;
until $\forall p$ *defined in* Q : $P = last\_P$

Fig. 1. Translation template 1.

4. $\mathtt{I}_3^1 = \emptyset$, $\mathtt{I}_3^2 = \{\mathtt{all}_3(\mathtt{a}), \mathtt{all}_3(\mathtt{b}), \mathtt{all}_3(\mathtt{d}), \mathtt{delta}_3(\mathtt{e}), \mathtt{all}_3(\mathtt{e})\}$
5. $I_j = \emptyset$ for $j > 3$.

By Theorem 1, we conclude that there are two stable models for the program: $I^1 = I_0 \cup I_1 \cup I_2^1$ and $I^2 = I_0 \cup I_1 \cup I_2^2 \cup I_3^2$. Clearly, any realistic implementation, such as that provided in $\mathcal{LDL}++$, computes nondeterministically only *one* of the possible stable models.

4 An Operational Semantics for Datalog++

We now translate the iterated stable model procedure into a more concrete form by using relational algebra operations and control constructs. Following the style of [24], we associate with each predicate p a relation P—same name capitalized.

The elementary deduction step $T_Q(I)$ is translated as an assignment to appropriate relations:

$$I' := T_Q(I) \qquad \longleftrightarrow \qquad \forall p \in def(Q)\ P := EVAL(p, Rels),$$

where $Rels = \{P |\ p \in def(Q)\} \cup \{R |\ R \in EDB\}$, i.e., the relations defined in the clique Q together with the extensional relations, and $EVAL(p, Rels)$ denotes, in the notation of [24], a single evaluation step of the rules for predicate p with respect to the current extension of relations in $Rels$.

We show the translation of Datalog++ cliques incrementally in three steps, starting with simple Datalog programs and stratified negation, then introducing the Choice construct and eventually describing how to translate the full language. The translation of a whole program can be trivially obtained by gathering the single translated cliques in the natural order.

4.1 Datalog with Stratified Negation

We can straightforwardly apply the transformation given in [24] for safe stratified Datalog programs, where each negative literal referring to a previously computed or extensional relation is translated to the complement of the relation w.r.t. the universe of constants (see Fig. 1).

The translation is illustrated in the following:

Example 4. The program:

$$\mathtt{p}(\mathtt{nil}, \mathtt{a}).$$
$$\mathtt{p}(\mathtt{X}, \mathtt{Y}) \leftarrow \mathtt{p}(\mathtt{X}, \mathtt{Z}), \mathtt{arc}(\mathtt{Z}, \mathtt{Y}), \neg\mathtt{bad\_node}(\mathtt{Y}).$$

is translated to the following naive evaluation procedure, where $EVAL(\ldots)$ is instanced with an appropriate RA query:

$P := \emptyset$;
repeat
  $last\_P := P$;
  $P := \{< nil, a >\}$
  $\cup\, \pi_{X,Y}\Big(last\_P(X, Z) \bowtie Arc(Z, Y) \bowtie \overline{Bad\_node(Y)}\Big)$;
until $P = last\_P$.

4.2 Adding Choice

Now, we need to translate, in relational algebra terms, the operations which compose a nondeterministic computation. Following the approach of [13], [12], we partition the rules of $SV(Q)$ (the stable version of Q) into three sets:

$\mathcal{C} = chosen$ rules
$\mathcal{D} = diffChoice$ rules
$\mathcal{O} = SV(Q) \setminus (\mathcal{C} \cup \mathcal{D})$ (i.e., the remaining rules).

Now, the nondeterministic fixpoint procedure which computes the stable models of a choice program is represented by Fig. 2.

At Step 1, the procedure tries to derive all possible atoms from the already given choices (none at the first iteration), so that, at Step 2, it can collect all candidate atoms which can be chosen later. If there is any such atom (i.e., we have not reached the fixpoint of the evaluation), then we can nondeterministically choose one of them (Step 4) and then propagate the effects of such choice (Step 5) in order to force the FD which it implies. We are then ready to repeat the process.

Example 5. The stable version of the students-advisors example seen in Section 2 is the following:

$$\mathcal{O} : \mathtt{st\_ad}(\mathtt{St}, \mathtt{Ad}) \leftarrow \mathtt{major}(\mathtt{St}, \mathtt{Area}), \mathtt{faculty}(\mathtt{Ad}, \mathtt{Area}), \mathtt{chosen}(\mathtt{St}, \mathtt{Ad}).$$
$$\mathcal{C} : \mathtt{chosen}(\mathtt{St}, \mathtt{Ad}) \leftarrow \mathtt{major}(\mathtt{St}, \mathtt{Area}), \mathtt{faculty}(\mathtt{Ad}, \mathtt{Area}), \neg\mathtt{diffchoice}(\mathtt{St}, \mathtt{Ad}).$$
$$\mathcal{D} : \mathtt{diffchoice}(\mathtt{St}, \mathtt{Ad}) \leftarrow \mathtt{chosen}(\mathtt{St}, \mathtt{Ad}'), \mathtt{Ad} \neq \mathtt{Ad}'.$$

Following the above translation schema, then, we obtain the following procedure:

0. $St\_ad := \emptyset; Chosen := \emptyset; Diffchoice := \emptyset$;
1. $St\_ad := \pi_{St,Ad}(Major(St, Area) \bowtie Faculty(Ad, Area) \bowtie Chosen(St, Ad))$;
2. $Chosen' := \pi_{St,Ad}(Major(St, Area) \bowtie Faculty(Ad, Area) \bowtie \overline{Diffchoice(St, Ad)}\Big)$;
3. if $Chosen' = \emptyset$ then stop;
4b. Choose (fairly) $< st, ad > \in Chosen'$;
4c. $Chosen := Chosen \cup \{< st, ad >\}$;
5. $Diffchoice := Diffchoice \cup \Big(\{< st >\} \times \overline{\{< ad >\}}\Big)$;
6. goto 1

Notice that some steps have been slightly modified: 1) Step 1 has been simplified since the only rule in $\mathcal{O}$ is

0. **Init**
   $\forall p$ *defined in* $SV(Q) : P := \emptyset$
1. **Saturation**
   repeat
      $\forall p$ *defined in* $\mathcal{O} : last\_P := P$
      $P := EVAL(p, Rels)$
   until $\forall p$ *defined in* $\mathcal{O} : last\_P = P$
2. **Gather choices**
   $\forall chosen_r$ *defined in* $\mathcal{C} : Chosen'_r := EVAL(chosen_r, Rels)$
3. **Termination test**
   if $\forall chosen_r$ *defined in* $\mathcal{C} : Chosen'_r = \emptyset$ *then stop*
4. **Choice**
   Execute (fairly) the following
   a. *Choose* $Chosen'_r \neq \emptyset$
   b. *Choose* $\overline{t} \in Chosen'_r$
   c. $Chosen_r := Chosen_r \cup \{\overline{t}\}$
5. **Inhibit other choices**
   $\forall diffchoice_r$ *defined in* $\mathcal{D} : Diffchoice_r := EVAL(diffchoice_r, Rels)$
6. *goto 1*

Fig. 2. Translation template 2.

not recursive (modulo the chosen predicate), 2) here, we have only one choice rule, so Step 4a becomes useless and then it has been ignored, and 3) Step 5 is rewritten in a brief and more readable form, which has exactly the same meaning of that shown in the above general schema.

4.3 Adding XY–Stratification

Analyzing the evaluation procedure of an XY-clique Q, it is easy to see that, at each step n, the only atoms which can be derived are of the form $p(n, \ldots)$, i.e., all with the same stage argument and, then, the syntactic form of the rules ensures that such rules refer only to atoms $p(n, \ldots)$ and $p(n-1, \ldots)$. Then, the stage arguments in each rule serve only to distinguish the literals computed in the actual stage I from those computed in the previous stage $I-1$.

Therefore, we can safely omit the stage argument from each XY-recursive predicate, renaming the literals referring to a previous stage (i.e., those having stage I inside a rule with head having stage $I+1$) by adding the prefix "old_". This does not apply to exit–rules in which the stage argument value is significant and then must be preserved. We denote by Q' the resulting rules and by p' the predicate obtained from each p.

Example 6. The program Q

$$\mathtt{p}(7, \mathtt{a}).$$
$$\mathtt{p}(\mathtt{s}(\mathtt{I}), \mathtt{X}) \leftarrow \mathtt{q}(\mathtt{s}(\mathtt{I}), \mathtt{X}, \mathtt{Y}), \mathtt{r}(\mathtt{I}, \mathtt{X})$$

is translated into the new program Q'

$$\mathtt{p}(7, \mathtt{a}).$$
$$\mathtt{p}(\mathtt{X}) \leftarrow \mathtt{q}(\mathtt{X}, \mathtt{Y}), \mathtt{old\_r}(\mathtt{X}).$$

Now, it suffices to store in an external register J the value of the stage under evaluation. We can 1) fire the exit–rules having the same stage argument as J and, then, 2) evaluate the new rules in Q' (which are now stratified and possibly with choice) as described in the last two sections. When we have completely evaluated the actual stage, we need to store the newly derived atoms p in the corresponding old_p, to increment J and then to repeat the process in order to evaluate the next stage. The resulting procedure is presented in Fig. 3.

Here, we simply reduce the evaluation of the XY–clique to the iterated evaluation of its *stage instances* (Step 2) in a sequential ascending order (Step 4). Each stage instance is stratified modulo choice and then it can be broken into subcliques (Step 2) which can be translated by template 1 (if choice-free) or template 2 (if with choice). The resulting relations (Step 3) can be easily obtained by collecting, at each stage J, the relations P' and translating them into the corresponding P, i.e., adding to them the stage argument J.

Example 7. Let $g/2$ be an extensional predicate representing the edges of a graph. Consider the following clique Q:

0. **Init**
   $J := 0;$
   $\forall p'$ *defined in* $Q' : old\_P' := \emptyset$
   $P' := \emptyset$
1. **Fire exit rules**
   $\forall$*exit-rule* $r = p(j, \ldots) \leftarrow \ldots :$
      if $J = j$ *then* $P' := EVAL_r(p', Rels)$
2. **Fire Q'**
   Translate Q' *following the translation templates 1 and 2*
3. **Update relations**
   $\forall p'$ *defined in* $Q' : old\_P' := P'$
   $P := P \cup \{\langle J \rangle\} \times P';$
   $P' := \emptyset$
4. $J := J + 1;$ *goto 1*

Fig. 3. Translation template 3.

$$\Delta(\mathtt{nil},\mathtt{Y}) \leftarrow \mathtt{g}(\mathtt{a},\mathtt{Y}).$$
$$\Delta(\mathtt{s}(\mathtt{I}),\mathtt{Y}) \leftarrow \Delta(\mathtt{I},\mathtt{X}),\mathtt{g}(\mathtt{X},\mathtt{Y}),\neg\mathtt{all}(\mathtt{I},\mathtt{Y}).$$
$$\mathtt{all}(\mathtt{s}(\mathtt{I}),\mathtt{Y}) \leftarrow \mathtt{all}(\mathtt{I},\mathtt{Y}),\Delta(\mathtt{s}(\mathtt{I}),\_).$$
$$\mathtt{all}(\mathtt{I},\mathtt{Y}) \leftarrow \Delta(\mathtt{I},\mathtt{Y}).$$

The corresponding transformed clique Q' is:

$$r_0:\ \Delta(\mathtt{nil},\mathtt{Y}) \leftarrow \mathtt{g}(\mathtt{a},\mathtt{Y}). \qquad \text{(exit rule)}$$
$$r_1:\ \Delta'(\mathtt{Y}) \leftarrow \mathtt{old\_}\Delta'(\mathtt{X}),\mathtt{g}(\mathtt{X},\mathtt{Y}),\neg\mathtt{old\_all}'(\mathtt{Y}).$$
$$r_2:\ \mathtt{all}'(\mathtt{Y}) \leftarrow \mathtt{old\_all}'(\mathtt{Y}),\Delta'(\_).$$
$$r_3:\ \mathtt{all}'(\mathtt{Y}) \leftarrow \Delta'(\mathtt{Y}).$$

Q' can be partitioned into: *exit-rule* r_0, subclique $Q'_1 = \{r_1\}$, and subclique $Q'_2 = \{r_2, r_3\}$. Applying translation template 3, we obtain:

0. $J := 0; old\_\Delta' := \emptyset; \Delta' := \emptyset;$
   $old\_All' := \emptyset; All' := \emptyset;$
1. if $J = 0$ then $\Delta' := \pi_Y(\sigma_{X=a}(G(X,Y)))$;
2a. $\Delta' := \Delta' \cup \pi_Y(old\_\Delta'(X) \bowtie G(X,Y) \bowtie \overline{All'(Y)})$;
2b. $All' := All' \cup [\pi_Y(old\_All'(Y) \bowtie \Delta') \cup \Delta']$;
3. $old\_\Delta' := \Delta'; \Delta := \{< J >\} \times \Delta'; \Delta' := \emptyset;$
   $old\_All' := All'; All := \{< J >\} \times All'; All' := \emptyset;$
4. $J := J + 1$; goto 1

Notice that Steps 2a and 2b have been simplified w.r.t. translation template 1 because Q' is not recursive and then the iteration cycle is useless (indeed, it would reach saturation on the first step and then exit on the second one).

5 Optimization of Datalog++ Queries

A systematic study of query optimization techniques is realizable on the basis of the concrete implementation of the iterated stable model procedure discussed in the previous section. We now sketch a repertoire of ad hoc optimizations for Datalog++ by exploiting the particular syntactic structure of programs and queries and the way they use the temporal arguments.

First of all, we observe that the computations of translation template 3 never terminate. An obvious termination condition is to check that the relations computed at two consecutive stages are empty. To this purpose, translation template 3 can be modified by inserting the following instruction between Step 2 and 3:

$$\text{if } \forall p \text{ defined in } Q:\ P = old\_P = \emptyset \text{ then stop}.$$

A more general termination condition is applicable to *deterministic* cliques under the assumption that the external calls to the predicates of the clique do not specify particular stages, i.e., external calls are of the form $p(\_,\ldots)$. In this case, the termination condition above can be simplified as follows:

$$\text{if } \forall p \text{ defined in } Q:\ P = old\_P \text{ then stop}.$$

5.1 Forgetful-Fixpoint Computations

In many applications (e.g., modeling updates and active rules [25], [10]), queries are issued with reference to the final stage only (which represents the commit state of the database). Such queries often exhibit the form

$$\mathtt{p}(\mathtt{I},\mathtt{X}),\neg\mathtt{p}(\mathtt{s}(\mathtt{I}),\_),$$

with the intended meaning "find the value X of p in the final state of p." This implies that 1) when computing the next stage, we can forget all the preceding states but the last one (see [26]) and 2) if a stage I such that $\mathtt{p}(\mathtt{I},\mathtt{X}),\neg\mathtt{p}(\mathtt{s}(\mathtt{I}),\_)$ is unique, we can quit the computation process once the above query is satisfied. For instance, the program in Example 7 with the query $\Delta(I,X),\neg\Delta(s(I),\_)$ computes the leaf nodes at maximal depth in a breadth-first visit of the graph rooted in a. To the purpose of evaluating this query, it suffices to 1) keep track of the last computed stage only and 2) exit when the current Δ is empty. The code for the program of Example 7 is then optimized by:

1. Replacing step 3 with:
   $$3.\ old\_\Delta' := \Delta'; \Delta' := \emptyset;$$
   $$old\_All' := All'; All' := \emptyset;$$
   i.e., dropping the instructions that record previous stages;
2. Insert between steps 2 and 3 the instruction:
   $$\text{if } \Delta' = \emptyset \text{ then } \Delta := \{< J-1 >\} \times old\_\Delta'; \text{ stop}.$$

Another interesting case occurs when the answer to the query is distributed along the stages, e.g., when we are interested in the answer to a query such as $\Delta(\_,\mathtt{X})$, which ignores the stage argument. In this case, we can collect the partial answers via a gathering predicate defined with a copy-rule. For instance, the all predicate in Example 7 collects all the nodes reachable from a. Then, the query $\mathtt{all}(\mathtt{I},\mathtt{X}),\neg\mathtt{all}(\mathtt{s}(\mathtt{I}),\_)$, which is amenable for the described optimization, is equivalent to the query $\Delta(\_,\mathtt{X})$, which, on the contrary, does not allow it. Therefore, by (possibly) modifying the program with copy-rules for the all predicate, we can systematically apply the space optimized forgetful-fixpoint.

5.2 Delta-Fixpoint Computations

We already mentioned the presence of a *copy-rule* in Example 7:

$$\mathtt{all}(\mathtt{s}(\mathtt{I}),\mathtt{X}) \leftarrow \mathtt{all}(\mathtt{I},\mathtt{X}),\Delta(\mathtt{s}(\mathtt{I}),\_).$$

Its effect is that of copying all the tuples from the stage I to the next one, if any. We can avoid such useless space occupation by maintaining, for each stage, only the modifications which are to be applied to the original relation in order to obtain the actual version. For example, the above rule represents no modification at all and, hence, it should not have any effect; indeed, it suffices to keep track of the additions to the original database requested by the other rule:

$$\mathtt{all}(\mathtt{I},\mathtt{X}) \leftarrow \Delta(\mathtt{I},\mathtt{X}),$$

which can be realized by a supplementary relation $\texttt{all}^+$ containing, at each stage, the new tuples produced. In the case that we replace the *copy-rule* with a *delete-rule* of the form:

$$\texttt{all(s(I), X)} \leftarrow \texttt{all(I, X)}, \Delta\texttt{(s(I), \_)}, \neg\texttt{q(X)},$$

we need simply to keep track of the negative contribution due to literal $\neg\texttt{q(X)}$, which can be stored in a relation $\texttt{all}^-$. Each $\texttt{all(I, \ldots)}$ can then be obtained by integrating $\texttt{all(0, \ldots)}$ with all the $\texttt{all}^+\texttt{(J, \ldots)}$ and $\texttt{all}^-\texttt{(J, \ldots)}$ atoms, with $J \leq I$. This method is particularly effective when $\texttt{all(0, \ldots)}$ is a large relation. To illustrate this point, let us assume that the program of Example 7 is modified by adding a new exit rule for relation all:

$$\texttt{all(0, X)} \leftarrow \texttt{r(X)}.$$

where r is an extensional predicate. The resulting code is then the following:

0. $J := 0$; $old\_\Delta' := \emptyset$; $\Delta' := \emptyset$;
   $old\_All^+ := \emptyset$; $All^+ := \emptyset$;
1. if $J = 0$ then $\Delta' := \pi_Y(\sigma_{X=a}(G(X, Y)))$;
2a. $\Delta' := \Delta' \cup \pi_Y(old\_\Delta'(X) \bowtie G(X, Y)$
   $\bowtie \overline{r(Y) \cup old\_All^+(Y)})$;
2b. $All^+ := \Delta' \cup old\_All^+$;
3. $old\_\Delta' := \Delta'$; $\Delta := \{< J >\} \times \Delta'$; $\Delta' := \emptyset$;
   $old\_All' := All'$; $All := \{< J >\} \times All'$; $All' := \emptyset$;
4. $J := J + 1$; goto 1

In this way, we avoid the construction of relation All, i.e., the replication of relation r at each stage. In fact, All is reconstructed on the fly when needed (Step 2a).

5.3 Side-Effect Computations

A direct combination of the previous two techniques gives rise to a form of *side-effect* computation. Let us consider, as an example, the nondeterministic ordering of an array performed by swapping at each step any two elements which violate ordering. Here, the array $\texttt{a} =< \texttt{a}_1, \cdots, \texttt{a}_\texttt{n} >$ is represented by the relation a with extension $\texttt{a}(1, a_1), \cdots, \texttt{a}(n, a_\texttt{n})$.

```
ar(0,P,Y)              ←a(P,Y).
swp(I,P1,P2)           ←ar(I,P1,X),ar(I,P2,Y), X>Y,P1<P2,
                         choice((I),(P1,P2)).
ar(s(I),P,X)           ←ar(I,P,X), ¬swp(I,P,_),swp(I,_,P),swp(I,_,_).
ar(s(I),P,X)           ←ar(I,P1,X), swp(I,P1,P).
ar(s(I),P,X)           ←ar(I,P1,X), swp(I,P,P1).
? ar(I,X,Y),¬ar(s(I),_,_)
```

At each stage i, we nondeterministically select an unordered pair x, y of elements, delete the array atoms $\texttt{ar(i, p1, x)}$ and $\texttt{ar(i, p2, y)}$ where they appear, and add the new atoms $\texttt{ar(s(i), p1, y)}$ and $\texttt{ar(s(i), p2, x)}$ representing the swapped pair. The query allows a forgetful-fixpoint computation (in particular, stage selected by the query is unique) and the definition of predicate ar is composed by delete-rules and an add-rules. This means that, at each step, we can 1) forget the previously computed stages (but the last) and 2) avoid copying most of relation ar, keeping track only of the deletions and additions to be performed. If the requested updates are immediately performed, the execution of the proposed program then boils down to the efficient iterative computation of the following (nondeterministic) Pascal-like program:

$$\textbf{while } \exists I\ a[I] > a[I+1] \textbf{ do } swap(a[I], a[I+1]) \textbf{ od}.$$

6 Conclusions

The work reported in this paper concerning fixpoint/operational semantics and the optimization of a logic database language for nondeterministic and nonmonotonic reasoning constitutes the starting point for an actual implemented system. Such a project is currently in progress, on the basis of the $\mathcal{LDL}++$ system developed at UCLA. We plan to incorporate the proposed optimization into the $\mathcal{LDL}++$ compiler to the purpose of:

- Evaluating how *effective* the proposed optimizations are for realistic $\mathcal{LDL}++$ programs, i.e., whether they yield better performance or not,
- Evaluating how *applicable* the proposed optimizations are for realistic $\mathcal{LDL}++$ programs, i.e., how often they can be applied, and
- Experimenting with the integration of the proposed optimizations with the classical optimization techniques, such as magic-sets.

Acknowledgments

This paper is a revised, extended version of two extended abstracts [9], [14].

References

[1] K.R. Apt, H. Blair, and A. Walker, "Towards a Theory of Declaritive Knowledge," *Proc. Workshop Foundations of Deductive Databases and Logic Programming,* J. Minker, ed., pp. 89-148, 1988.

[2] N. Arni, K. Ong, S. Tsur, and C. Zaniolo, "$\mathcal{LDL}$++: A Second Generation Deductive Databases Systems," technical report, MCC Co., 1993.

[3] F. Bonchi, F. Giannotti, G. Mainetto, and D. Pedreschi, "A Classification-Based Methodology for Planning Audit Strategies in Fraud Detection," *Proc. Fifth ACM-SIGKDD Int'l Conf. Knowledge Discovery & Data Mining (KDD'99),* pp. 175-184, Aug. 1999.

[4] A. Brogi, V.S. Subrahmanian, and C. Zaniolo, "The Logic of Totally and Partially Ordered Plans: A Deductive Database Approach," *Annals Math. in Artificial Intelligence,* vol. 19, pp. 59-96, 1997.

[5] J. Chomicki, "Temporal Deductive Databases," *Temporal Databases: Theory, Design and Implementation,* A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, eds., pp. 294-320, 1993.

[6] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming," *Proc. Fifth Int'l Conf. Logic Programming,* pp. 1070-1080, 1988.

[7] F. Giannotti, S. Greco, D. Saccà, and C. Zaniolo, "Programming with Non Determinism in Deductive Databases," *Annals Math. in Artificial Intelligence,* vol. 19, pp. 97-125, 1997.

[8] F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi, "Datalog++: A Basis for Active Object-Oriented Databases," *Proc. DOOD '97, Fifth Int'l Conf. Deductive and Object-Oriented Databases,* pp. 283-301, 1997.

[9] F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi, "Query Answering in Non Deterministic Non Monotonic Logic Databases," *Proc. Conf. Flexible Query-Answering Systems (FQAS '98),* T. Andreasen, H. Christiansen, and H. Larsen, eds., 1998.

[10] F. Giannotti, G. Manco, and D. Pedreschi, "A Deductive Data Model for Representing and Querying Semistructured Data," *Proc. Second Int'l Workshop Logic Programming Tools for Internet Applications,* July 1997.
[11] F. Giannotti, G. Manco, D. Pedreschi, and F. Turini, "Experiences with a Logic-Based Knowledge Discovery Support Environment," *Proc. 1999 ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery (SIGMOD '99 DMKD),* 1999.
[12] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo, "Non-Determinism in Deductive Databases," *Proc. Second Int'l Conf. Deductive and Object-Oriented Databases (DOOD '91),* pp. 129-146, 1991.
[13] F. Giannotti, D. Pedreschi, and C. Zaniolo, "Semantics and Expressive Power of Non Deterministic Constructs for Deductive Databases," *J. Computer and Systems Sciences,* 1999.
[14] F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi, "On the Effective Semantics of Temporal, Non-Monotonic, Non-Deterministic Logic Languages," *Proc. Int'l Conf. Computer Science Logic (CSL '98),* pp. 58-72, Aug. 1998.
[15] F. Giannotti, G. Manco, M. Nanni, D. Pedreschi, and F. Turini, "Integration of Deduction and Induction for Mining Supermarket Sales Data," *Proc. Third Int'l Conf. Practical Applications of Knowledge Discovery and Data Mining (PADD '99),* pp. 79-94, Sept. 1999.
[16] D. Kemp, K. Ramamohanarao, and P. Stuckey, "ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS," *Proc. Fourth Int'l Conf. Deductive and Object-Oriented Databases (DOOD '95),* pp. 91-108, 1995.
[17] R. Krishnamurthy and S. Naqvi, "Non-Deterministic Choice in Datalog," *Proc. Third Int'l Conf. Data and Knowledge Bases,* pp. 416-424, 1988.
[18] B. Lüdascher, U. Hamann, and G. Lausen, "A Logical Framework for Active Rules" *Proc. Seventh COMAD Int'l Conf. Management of Data,* 1995.
[19] M. Nanni, "Nondeterminism and XY-Stratification in Deductive Databases (in Italian)," master's thesis, Dept. of Computer Science, Univ. of Pisa, 1997.
[20] S. Naqvi and S. Tsur, *A Logic Language for Data and Knowledge Bases.* Computer Science Press, 1989.
[21] T.C. Przymusinski, "Every Logic Program has a Natural Stratification and an Iterated Fix Point Model," *Proc. Eighth ACM PODS Symp. Principles of Database Systems,* pp. 11-21, 1989.
[22] K.A. Ross, "Modular Stratification and Magic Sets for Datalog Program with Negation," *J. ACM,* vol. 41, no. 6, pp. 1216-1266, Nov. 1994.
[23] D. Saccà and C. Zaniolo, "Stable Models and Non-Determinism in Logic Programs with Negation," *Proc. ACM Symp. Principles of Database Systems,* pp. 205-217, 1990.
[24] J.D. Ullman, *Principles of Database and Knowledge-Base Systems,* volume I-II. Computer Science Press, 1989.
[25] C. Zaniolo, "Active Database Rules with Transaction Conscious Stable Model Semantics," *Proc. Fourth Int'l Conf. Deductive and Object-Oriented Databases (DOOD '95),* pp. 55-72, 1995.
[26] C. Zaniolo, N. Arni, and K. Ong, "Negation and Aggregates in Recursive Rules: The $\mathcal{LDL}++$ Approach," *Proc. Third Int'l Conf. Deductive and Object-Oriented Databases (DOOD '93),* 1993.
[27] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R. Zicari, *Advanced Database Systems.* Morgan Kaufman, 1997.

Fosca Giannotti graduated in computer science (Laurea in Scienze dell' Informazione, summa cum laude) from the University of Pisa in 1982. Since then, she has been a research assistant in the CS Department of the University of Pisa, a researcher at the R&D labs of several Italian software companies, and a visiting scientist at The Microelectronics and Computer Technology Corporation (MCC), Austin, Texas, involved in the LDL (Logic Database Language) project. She is currently a senior researcher at CNUCE, Institute of CNR (Italian National Research Council) in Pisa. Her current research interests include knowledge discovery and data mining, spatio-temporal reasoning, and database programming languages design, implementation, and formal semantics, especially in the field of logic database languages. She is the coordinator of 1) Datasift, an Italian regional project aimed at developing a market basket analysis system for supermarket sales based on data mining and 2) DeduGIS, an Esprit working group funded by the European Commission, with 10 academic and industrial partners from four UE countries. She has taught classes on databases and datamining at universities in Italy and abroad, and has been on the PC of various conferences in the area of logic programming and databases, including EDBT 2000.

Giuseppe Manco graduated summa cum laude in computer science from the University of Pisa in December 1994. In 1997, he began a PhD fellowship in computer science in the Department of Computer Science of the University of Pisa. In 1998, he was a visiting fellow at the CWI Institute in Amsterdam (NL), where he worked on the KESO (Knowledge Extraction for Statistical Offices) project. He is currently a junior researcher at CNUCE (Institute of the National Research Council of Italy). His scientific interests are in logic and databases; in particular, his research activity covers integration of active, deductive and object-oriented features in databases, semistructured data management, data mining, and knowledge discovery.

Mirco Nanni holds a laurea degree in computer science (University of Pisa, 1997). Since 1998, he has been a PhD student in computer science at the University of Pisa. From April to June 1999, he was a visiting fellow at the University of Maryland, College Park, working on probabilistic agent programming. His current research interests are deductive databases, data mining and knowledge discovery, distributed interactive systems, and agent programming.

Dino Pedreschi received the PhD degree in computer science from the University of Pisa in 1987. He is currently an associate professor in the Dipartimento di Informatica of the University of Pisa, serving as the coordinator of undergraduate and graduate studies in computer science. He has been a visiting scientist and professor at the University of Texas at Austin (1989/1990), at CWI Amsterdam (1993), and at UCLA (1995). He has been the coordinator of the working group nondeterminism in deductive databases, jointly sponsored by the European Union and the US National Science Foundation, involving four European and two US universities and research centers. His current research interests are in logic in databases and, particularly, in data analysis, in deductive databases, in the integration of data mining and database querying, and in formal methods for deductive computing. He has taught classes on programming languages and databases at universities in Italy and abroad, and has participated on the PC of various conferences in the areas of logic programming and databases.